AD-A013 570

ADAPTIVE PRODUCTION SYSTEMS

D. A. Waterman

Carnegie-Mellon University

Prepared for:

Air Force Office of Scientific Research
Advanced Research Projects Agency
National Institutes of Health

December 1974

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER [illegible]TR-75-1031	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) ADAPTIVE PRODUCTION SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) D. A. Waterman	8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO-2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE December 1974
	13. NUMBER OF PAGES 72
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) 1400 Wilson Blvd. Arlington, VA 22209	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Adaptive production systems are defined and used to illustrate adaptive techniques in production system construction. A learning paradigm is described within the framework of adaptive production systems, and is applied to a simple rote learning task, a nonsense syllable association and discrimination task, and a serial pattern acquisition task. It is shown that with the appropriate production building mechanism, all three tasks can be solved using similar production system learning techniques. The adaptive production systems are quite parsimonious; that is, the learning program is represented in exactly the same fashion

continued

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ABSTRACT continued/Block 20
as the information being learned. Both are represented as production rules in a single production system. This eliminates the need for two types of control in the system; one for activating the learning mechanism and another for accessing the information learned. In other words, the concepts learned are not passive, static structures which must be given a special interpretation, but rather are self-contained programs which are executed automatically in the course of executing the learning mechanism.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ADAPTIVE PRODUCTION SYSTEMS

by D. A. Waterman

Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Complex Information Processing

Working Paper #285

December, 1974

DDC
RECEIVED
AUG 18 1975
B

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

This work was supported in part by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 monitored by the Air Force Office of Scientific Research and in part by the NIH Grant MH-07722.

ib

ABSTRACT

Adaptive production systems are defined and used to illustrate adaptive techniques in production system construction. A learning paradigm is described within the framework of adaptive production systems, and is applied to a simple rote learning task, a nonsense syllable association and discrimination task, and a serial pattern acquisition task. It is shown that with the appropriate production building mechanism, all three tasks can be solved using similar production system learning techniques.

The adaptive production systems are quite parsimonious; that is, the learning program is represented in exactly the same fashion as the information being learned. Both are represented as production rules in a single production system. This eliminates the need for two types of control in the system; one for activating the learning mechanism and another for accessing the information learned. In other words, the concepts learned are not passive, static structures which must be given a special interpretation, but rather are self-contained programs which are executed automatically in the course of executing the learning mechanism.

ADAPTIVE PRODUCTION SYSTEMS

by D. A. Waterman

This paper presents recent results in the design and use of adaptive or self-modifying production systems. A production system (Newell and Simon, 1972; Newell, 1973) can be thought of as simply a collection of production rules, that is, condition-action pairs, C => A, where the left side of each pair is a set of conditions relevant to the contents of a particular data base or working memory and the right side is a list of actions, each of which can modify the contents of that memory. The production systems to be discussed are written in PAS-II (Waterman and Newell, 1973; Waterman, 1973) and each is represented as a set of ordered production rules as illustrated below.

$$C_1 \Rightarrow A_1$$

$$C_2 \Rightarrow A_2$$

$$C_3 \Rightarrow A_3$$

The control cycle consists of selecting one production rule from the set and executing its actions. The first rule (from top to bottom) whose conditions match the working memory is the one selected. After the actions associated with the selected rule are executed the cycle starts again, from the top. This process continues until no conditions match.

An adaptive or self-modifying production system is defined to be one which, through its actions, can modify its own production rules. There are three principal ways such modification can take place: by adding new rules, by deleting old rules, and by changing existing rules. The adaptive production systems to be described in this paper use just one of these three modification techniques: addition of new rules. Thus the adaptive production systems not only contain actions which can modify the contents of working memory but also actions which can add new rules to the system.*

We will now postulate a common machinery for performing a variety of learning tasks. This machinery consists of (1) a production system interpreter for ordered production systems, (2) a production system representation for learning programs, (3) production rule actions capable of constructing and adding new rules to the system, and (4) the learning technique of adding new production rules above error-causing rules to correct the errors. Three types of learning tasks are investigated: arithmetic, verbal association, and series completion. The primary purpose of the investigation is to define and clarify the machinery needed for these tasks and show how it can be implemented within an adaptive production system framework.

The programs for all three tasks are written as short production systems which access a single working memory composed of an ordered set of memory elements. When production rules "fire," i.e., their actions are executed, they modify working memory by adding, deleting, or rearranging memory elements. Since rules fire only when their conditions match memory, such changes may cause different rules to fire and new memory modifications to be made. Thus the system cycles through various states, using working memory as a storage buffer for holding initialization data and intermediate results. Most of the actions are designed only to modify working memory, however a few are able to modify the production system itself by removing elements from working memory, assembling these elements into a production rule, and adding this production rule to the production system. These actions give the system its self-modification capability.

*The production rules are not considered part of the data base or memory.

The arithmetic learning task is relatively simple. It consists of learning to add or divide integers given only an ordering over the set of integers as initialization data. When the production system is given two integers to add, it uses the ordering to create a set of production rules which partially define the successor function for integers, and then uses this newly defined function to calculate the desired sum. After each sum is calculated, other production rules are added which effectively define the addition table for integers. Thus in the course of problem solving the system learns both the successor function and addition rules.

The verbal association task involves learning pairs of nonsense syllables, such that given the first syllable of the pair the system can respond with the second. The program which performs this task is essentially a production system implementation of EPAM (Feigenbaum, 1963). Instead of growing an EPAM discrimination net, the system creates a set of production rules which are equivalent to such a net. Two implementations of EPAM are discussed, an elementary one which exhibits stimulus generalization, and a more elaborate one which exhibits both stimulus and response generalization.

The series completion task consists of presenting a short sequence of symbols, such as AABBAABB, and then requesting a prediction of what symbols should come next in the sequence. The production systems for series completion create production rules which represent hypotheses about which symbol contexts lead to which new symbols, i.e., "two A's always lead to a B." These rules, taken together, constitute the concept of the series and are used by the production systems to predict new symbols. Two types of series completion production systems are discussed. The first handles simple series, that is, series which involve repetition and don't require knowledge about external alphabets; i.e., AMMAMMAMM. The second handles more complicated series which require the use of a next or prior relation on some alphabet, i.e., AABBCCDD.

The paper is organized as follows: section II contains a description of the PAS-II production system interpreter, section III describes adaptive production systems for arithmetic tasks, and section IV discusses production system simulations of EPAM. Section V describes production systems for series completion tasks, and section VI contains concluding remarks about adaptive production systems.

II. PAS-II PRODUCTION SYSTEM

PAS-II is an interactive information processing system designed to aid the user in analyzing verbal problem solving protocols. Within PAS-II the user may write production systems and have them executed by the PAS production system interpreter. This interpreter is modeled after PSG (Newell, 1972, 1973) and will interpret ordered production systems that access multiple data bases or memories.

Production rules in the PAS system consist of condition-action pairs, where the condition side is a set of conditions with implicit MEMBER and AND functions and the action side is an ordered list of independent actions. For example, a rule that would check memory STM to see if it contained both elements (A) and (B), and if so would deposit elements (C) and (D) into STM would be written:

(A) (B) => (DEP (C)) (DEP (D)), (1)

where the action DEP deposits its argument into memory. Also necessary is a definition of the initial contents of STM, such as:

1. STM = (A) (B).

All the production systems described in this paper access only a single memory, STM.

The control cycle of the PAS-II production system interpreter consists of two major mechanisms:

1. RECOGNIZE: a production rule (condition-action pair), whose conditions match (or satisfy) working memory is selected from the collection of rules. If no rule matches working memory the system halts.

2. ACT: the actions (right-hand side) of the selected rule are executed, modifying working memory.

A cycle in the system is defined to be a single RECOGNIZE-ACT sequence.

RECOGNIZE. The RECOGNIZE mechanism in the production system interpreter selects a rule to be executed. When more than one rule matches the working memory a conflict occurs, since RECOGNIZE must produce a single rule for the ACT mechanism to work on. Conflict resolution consists of applying some scheme which selects a particular rule from those that match memory. The only conflict resolution scheme used in the production systems in this paper is that of priority ordering. Thus the rule recognized is just the highest priority rule whose conditions match the data base.

The match mechanism in PAS assumes the conditions are in implicit MEMBER and AND notation and scans the condition elements in order from left to right checking to see if each element is in the working memory. When all the condition elements in a rule match corresponding elements in memory, the memory elements are automatically brought to the front of memory (just before actions are executed) in the order specified in the rule. A memory element can match only one condition element in any rule and the order of the memory elements does not have to correspond to the order of the condition elements. For example, the conditions (A) (B) (A) will match the memory STM: (B) (A) (A), but not the memory (A) (B).

A condition element will match a memory element if the memory element contains all the items the condition element contains, in the same order, starting from the beginning of the memory element. Thus condition element (A T) will match memory elements (A T), and (A T E) but not elements (A), (T A), or (T A T).

The match routine will search for the absence of a memory element if the condition element is preceeded by a minus sign (-). Thus the conditions (A) - (B) - (C) will match any memory which contains (A) but does not contain (B) and also does not contain (C).

Free variables* can be used in the condition elements and are denoted x1, x2, ..., xn. When a match occurs each item in the memory element which corresponds to a variable is bound to that variable. Then when a bound variable occurs in an action the value of the variable is used. For example, if we have memory STM: (A) (B (L)) and the rule:

```
(x1) (B x2)   =>    (DEP x2) ,
```

x1 will be bound to A, and x2 to (L). The action taken will be to deposit (L) into memory.

ACT. The ACT mechanism takes the rule specified by RECOGNIZE and executes all its actions, one at a time, in order going from left to right. Then the RECOGNIZE-ACT control cycle starts again and repeats until no rules match the data base.

The specification of actions in a production system is critical since to a large extent it determines the grain of the system. If the grain is too coarse the system still functions, but a single action may embody most of the interesting activity and thus obscure it from view. The criterion in defining actions seems to be to make the actions primitive enough so the trace of the production system will exhibit the activity deemed interesting.

The PAS Production system actions to be described are primitive ones and fall into two main categories: basic actions and modification actions. Only the actions used in the examples to be presented are discussed here. For the complete set of actions available see the PAS-II Reference Manual (Waterman, 1973).

*Variables which have restricted domains may also be defined (Waterman, 1973).

Basic Actions. The basic actions used in the examples are shown below with their definitions.

(DEP a): Deposit a into front of memory.

(REM a): Remove first occurrence of a from memory.

(REP a b n): Replace a with b in the nth element of memory.

(SAY a): Print a. Any number of arguments are permitted.

(CLEAR a): All elements in memory are cleared (erased) except a.

(ATTEND): Read from the terminal, permitting user to insert information into memory.

(STOP): Stop production system execution.

The actions assume that memory is an ordered list of elements going from left to right. Thus DEP places elements into memory at the left and REP counts elements starting from the left. If the third argument to REP is omitted, the interpretation is to replace a with b in the first element of memory.

Modification Actions. The modification actions are the embodiment of the mechanism which gives the production system its adaptive or self-modifying power. The total set of PAS modification actions are shown below with their definitions.

(COND a): Deposits (COND a) into memory and is exactly equivalent to (DEP (COND a)).

(ACTION a): Deposits (ACTION a) into memory and is exactly equivalent to (DEP (ACTION a)).

(MARK a): Marks each element in memory that just matched the condition elements on the left side of the rule containing (MARK a). An element e is marked by changing it to (a e).

(USED): This is a special case of MARK and is exactly equivalent to (MARK USED).

(OLD): This is another special case of MARK and is exactly equivalent to (MARK OLD).

(PROD a): Creates a production rule and inserts it into the production system. The rule is created from all memory elements marked (COND...) and (ACTION...). The new rule is then inserted just in front of the first production rule that contains, in either its condition or action, an element identical to any of the arguments of PROD. If no such rule is found, the new rule is not inserted. If PROD has no arguments the rule is inserted at the very beginning of the production system, if it has the argument END, it is inserted at the end of the system. In all cases all memory elements marked (COND...) and (ACTION...) are removed from memory.

The actions COND, ACTION, USED and OLD are clearly superfluous since the same effect can be obtained using DEP and MARK. They were implemented, however, to make the production rules more readable.

The actions MARK and PROD are illustrated by the following simple example. Assume the initial conditions shown below:

```
STM:  (B) (A) (C) (ACTION (SAY DONE) (STOP)) (COND (C))
PS:   1.  - (B) => (DEP (B))
      2.  (A) (B) => (MARK COND) (PROD (A))
      3.  (C) => (DEP (A))
```

When the production system PS is fired, rule 2 is the first to match STM. The action (MARK COND) marks (A) and (B) and memory becomes:

```
STM:  (COND (A)) (COND (B)) (C) (ACTION (SAY DONE) (STOP)) (COND (C)).
```

Then the action (PROD (A)) creates a production out of the elements marked COND and ACTION, removes these elements from STM, and puts the new production immediately above the first rule that contains (A), in this case rule 2. The resulting memory and production system are shown below.

```
STM:        (C)
PS:          1.  - (B) => (DEP (B))
           1.5.  (A) (B) (C) => (SAY DONE) (STOP)
             2.  (A) (B) => (MARK COND) (PROD (A))
             3.  (C) => (DEP (A))
```

After the insertion of rule 1.5 the production system execution continues, finally terminating with the firing of rule 1.5.

Special Actions. The actions defined thus far are considered primitive actions. It was necessary, however, to also define a small set of non-primitive, problem specific actions for use in the verbal association and series completion production systems. These were needed to make the production system traces reflect the activity deemed interesting. These special actions: SUCC, PERCEIVE, OBSERVE, OBSERVE1, OBSERVE2, and PRODS, are described in detail in the next three sections.

Predecessor and Successor Actions. In the PAS production system predecessors and successors on letters and integers can be accessed implicitly by placing apostrophes before or after variables in either the condition or action part of production rules. Thus the value of x1' is the successor of the value of x1, and the value of ''x1 is the double predecessor of the value of x1. The implicit predecessor and successor actions were not used in the arithmetic or verbal association production systems, but were needed in the more complex series completion systems. Also, the explicit successor action (SUCC) was implemented to increase efficiency. It changes every memory element with a number as the first item by replacing that number with its successor.

III. PRODUCTION SYSTEMS FOR ARITHMETIC TASKS

Two simple production systems will now be described, one for performing addition of integers and one for division. These systems were designed to illustrate adaptive production system techniques and to compare production system programming with more conventional programming methods. They do not attempt to model data on human performance in these tasks.

Addition. The production system for addition is shown in Figure 1. It consists of five production rules, the first two providing initialization, the next two performing the addition, and the last adding rules which define the successor function. The initialization rules fire only once, at the beginning of the execution of the production system.

```
1. (READY) (ORDER X1) => (REP (READY) (COUNT X1)) (ATTEND)
2. (N X1) - (NN) - (S NN) => (DEP (NN X1))
3. (COUNT X1) (M X1) (NN X2) (N X3) => (SAY X2 IS THE ANSWER)
   (COND (M X1) (N X3)) (ACTION (STOP))
   (ACTION (SAY X2 IS THE ANSWER)) (PROD) (STOP)
4. (COUNT) (NN) => (REP (COUNT) (S COUNT)) (REP (NN) (S NN) 2)
5. (ORDER X1 X2) => (REP (X1 X2) (X2)) (COND (S X3 X1))
   (ACTION (REP (S X3 X1) (X3 X2))) (PROD)
```

Figure 1. ADD: A Production System for Addition of Integers

When the action ATTEND is executed the system expects to be given two integers (a and b) in the form (M a) (N b). It then calculates a + b and prints the answer. The algorithm used to calculate the answer is illustrated by the program below:

```
2.1     add(m,n) = count ← o; nn ← n;
2.2     L1 if count = m then return(nn);          (2)
2.3        count ← s(count);
2.4        nn ← s(nn);
2.5        go(L1);
```

Here count and nn are local variables and s is the successor function. Count is initialized to zero and nn to n. Then count and nn are continuously incremented by one, using the successor function, until count equals m. The answer is then nn.

The ADD production system performs these steps with a few essential differences. First, it has no successor function, consequently it creates a production rule representation of the successor function by adding rules which tell it how to find the successor for particular integers. And second, once a sum is calculated it adds a rule that produces the answer directly the next time it is required. Thus it builds the addition table for integers.

There is a direct mapping, however, between the code in (2) and that in Figure 1. Rules 1 and 2 in Figure 1 correspond to line 2.1 of the above program. Rule 3 corresponds to 2.2, and rule 4 to 2.3 and 2.4 above. Rule 5 has no correspondent in (2) since the above code assumes the existence of the successor function, while the production system code creates it. Note also that 2.5 above, the GOTO statement, has no correspondent in Figure 1. In the ADD production system the function of the GOTO statement and label is handled by two mechanisms: control cycle repetition, which permits unlimited looping; and appropriate memory modification, which in this case makes rules 1 and 2 inoperative.

A trace of the ADD production system solving the problem 4 + 2 is shown in Appendix I. Memory is assumed to initially contain two things: a ready signal (READY) and an ordering over the set of integers (ORDER 0 1 2 3 4 5 6 7 8 9).* Notice that the system only adds successor rules when they are needed for the computation. The second time the system is given 4 + 2 it calculates it directly from the partial addition table just created. When given other problems, such as 2 + 1, the successor rules previously acquired are remembered and used.

Division. The production system for division is shown in Figure 2. This time there are six production rules; the first two provide initialization, the next three perform the division, and the last one creates successor and predecessor rules when needed.

As in the ADD system, the user gives the DIVIDE system two integers (a and b) in the form (M a) (N b), and the system calculates a ÷ b, including the remainder. This calculation is done according to the following program:

```
3.1  divide(m,n) = count ← o; ans ← o; remain ← m;
3.2  L1 if count = n then [ans ← s(ans); count ← o; remain ← m]
3.3                  else [if m = o then return(ans,remain);        (3)
3.4                        m ← p(m); count ← s(count)];
3.5     go(L1);
```

Here count, ans, and remain are local variables, s is the successor function, and p is the predecessor function. Basically, count is being incremented

*This ordering should extend to 18 for the system to handle addition of all single digit integers. It was kept to 9 for the sake of brevity in the trace.

```
1. (READY) => (REP (READY) (COUNT 0)) (DEP (ANS 0)) (ATTEND)
2. (M X1) - (REMAIN) => (DEP (REMAIN X1))
3. (ANS) (COUNT X1) (REMAIN X3) (N X1) (M X2) => (REP (ANS) (S ANS))
   (REP (X1) (0) 2) (REP (X3) (X2) 3)
4. (M 0) (ANS X1) (REMAIN X2) - (S) => (SAY X1 WITH REMAINDER X2) (STOP)
5. (M) (COUNT) - (S) => (REP (M) (P M)) (REP (COUNT) (S COUNT) 2)
6. (ORDER X1 X2) - (GO) => (REP (X1 X2) (X2)) (COND (P X3 X2))
   (ACTION (REP (P X3 X2) (X3 X1))) (PROD (GO)) (COND (S X3 X1))
   (ACTION (REP (S X3 X1) (X3 X2))) (PROD (READY))
```

Figure 2. DIVIDE: A Production System for Division of Integers

by one while m is being decremented by one until count = n. Then ans is incremented, count reset to zero and the process continues until m = o.

The DIVIDE production system uses the above algorithm but has no successor or predecessor functions. Instead it creates production rule representations of these functions by adding the appropriate rules when needed. As before, there is a direct mapping between the DIVIDE production system code and the LISP-like code shown in (3) above. The initialization rules, 1 and 2 in Figure 2, correspond to 3.1 above. Rule 3 corresponds to 3.2, 4 to 3.3, and 5 to 3.4. Again, the production building rule, 6, has no correspondent in (3) since the above program creates no new code. Note that the if-then-else statement in (3) is quite easily mapped into the DIVIDE production system because of the priority ordering of the production system rules. A trace of the DIVIDE system on the problem 3 ÷ 2 is shown in Appendix II. Again memory initially contains a ready signal and an ordering over the set of integers. To obtain the correct answer "1 with remainder 1" the system added the predecessor and successor rules shown.

The production systems for arithmetic are self-modifying but not really adaptive in the strict sense of the word. This is because they create new rules, not on the basis of external feedback, but rather on the basis of internal information, i.e., the ordering on the set of integers. Furthermore rules are added only when needed to solve the problem at hand. This is a good example of an explicit view of predetermined developmental potential. The system has the capacity to develop the addition table or the successor function on integers but does so only when the environment demands it.

IV. PRODUCTION SYSTEM IMPLEMENTATIONS OF EPAM

EPAM (Feigenbaum, 1963; Feigenbaum and Simon, 1964) is a computer program which simulates verbal learning behavior by memorizing three-letter nonsense syllables presented in associate pairs or serial lists. The program learns to predict the correct response when given a stimulus syllable by growing a discrimination net composed of nodes which are tests on the values of certain attributes of the letters in the nonsense syllables. Responses are stored at the terminal nodes, and are retrieved by sorting the stimuli down the net. A typical paired associate training sequence for this verbal learning task is shown in Figure 3.

Two production system implementations of EPAM will now be discussed, each using only the actions described in Section II plus a special action called PERCEIVE. This is a problem-specific action which recognizes the type and location of individual letters in a syllable, as shown below.

(PERCEIVE _a_ _b_):	Breaks syllable _a_ into individual letters and tags these letters with _b_ and a number specifying their order in the syllable.

For example, the action (PERCEIVE PAX ?) when executed adds the elements (1 P ?) (3 X ?) (2 A ?) to memory, indicating that the first letter in the syllable was P, the third X, and the second A. The elements are arranged first, third, second, to reflect the serial position effect in verbal learning (Deese and Kaufman, 1957).

The purpose of presenting the production system implementations of EPAM is to show the relation between discrimination nets and ordered production rules and to demonstrate the use of the modification actions in a verbal learning situation.

Stimulus	Response
PAX	CON
BEK	LUQ
CIT	DER
BUK	MAB
NAL	LEQ
REB	MOL
NOJ	PED

Figure 3. Paired Associate Training Sequence for Verbal Learning Task.

EPAM1. The first implementation, called EPAM1, is shown in Figure 4. Rules 1 and 2 are initialization rules only, the response and learning mechanisms are embodied in the last 5 rules. This simple version of EPAM grows a production system which is analogous to a discrimination net with tests for stimulus letters (i.e., "is the 3rd letter R?") at the intermediate nodes and complete responses at the terminal nodes. Consequently, the system can exhibit retroactive inhibition and stimulus generalization but not response generalization (Feigenbaum. 1963).

The operation of EPAM1 will be described by using an annotated listing (shown below) of the trace of the program learning two pairs of syllables: PAX-CON and PUM-JES. The program output is in upper case, the user input, after the ATTEND function, is in lower case.

```
*fire
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pax))
     STM: (STIM PAX) (READY)

     1 TRUE IN PS
     STM: (1 P ?) (3 X ?) (2 A ?) (STIM PAX)

     6 TRUE IN PS

?

OUTPUT FOR (ATTEND RESP) = (dep (resp con))
     STM: (RESP CON) (REPLY ?) (1 P ?) (3 X ?) (2 A ?)
        (STIM PAX)
```

Initially STM contains the signal (READY), which causes rule 2 to fire and the system to ask for the stimulus. Once the stimulus (STIM PAX) is obtained, rule 1 fires, adding the perceived stimulus components to memory as shown above. Now rule 6 is the first to match STM. As it fires it prints a question mark as the system's reply to the stimulus (i.e., it doesn't have anything associated with the syllable PAX), adds this reply to memory, and asks for the correct response.

```
1. (READY) (STIM X1)  =>  (REM (READY)) (PERCEIVE X1 ?)
2. (READY)  =>  (ATTEND STIM)
3. (REPLY) - (RESP)  =>  (ATTEND RESP)
4. (REPLY X1) - (RESP X1)  =>  (REP REPLY WRONG)
5. (USED X1) (WRONG X2)  =>  (REP USED COND)
6. - (RESP)  =>  (DEP (REPLY ?)) (SAY ?) (ATTEND RESP)
7. (X1 X2 ?) (RESP X3) (WRONG X4)  =>  (COND (X1 X2 ?))
   (ACTION (USED) (DEP (REPLY X3)) (SAY X3)) (PROD (SAY X4)) (STOP)
```

Figure 4. EPAM1: A Production System Implementation of EPAM

```
    4 TRUE IN PS
    STM: (WRONG ?) (RESP CON) (1 P ?) (3 X ?) (2 A ?)
       (STIM PAX)

    7 TRUE IN PS
NOW INSERTING
(1 P ?)  => (USED) (DEP (REPLY CON)) (SAY CON)
ON LINE  5.5
    STM: (1 P ?) (RESP CON) (WRONG ?) (3 X ?) (2 A ?)
       (STIM PAX)
```

Since there is now a reply in memory (?) that does not match the correct response (CON) rule 4 matches and changes the label REPLY to WRONG. Finally rule 7 matches memory with the variables bound as follows: x1=1, x2=P, x3=CON, and x4=?. Thus when 7 is fired it adds (COND (1 P ?)) and (ACTION (USED) (DEP (REPLY CON)) (SAY CON)) to memory, removes them from memory to create a new rule, and inserts that rule in front of the first rule that contains (SAY ?), in this case rule 6.

```
*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pum))
    STM: (STIM PUM) (READY)

    1 TRUE IN PS
    STM: (1 P ?) (3 M ?) (2 U ?) (STIM PUM)

  5.5 TRUE IN PS

CON

     STM: (REPLY CON) (USED (1 P ?)) (3 M ?) (2 U ?)
        (STIM PUM)
```

Before the second pair of syllables is presented, memory is initialized back to its original contents: (READY), and the system is fired. Again rules 2 and 1 match and are fired in the process of obtaining and perceiving the stimulus. But now the new rule, 5.5, matches memory and causes (1 P ?) to be marked USED, and the system to reply CON and add the reply to memory. This is an example of stimulus generalization: the system confused PUM with

PAX since it was only noticing first letters.

```
    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp ies))
    STM: (RESP JES) (REPLY CON) (USED (1 P ?)) (3 M ?)
       (2 U ?) (STIM PUM)

    4 TRUE IN PS
    STM: (WRONG CON) (RESP JES) (USED (1 P ?)) (3 M ?)
       (2 U ?) (STIM PUM)

    5 TRUE IN PS
    STM: (COND (1 P ?)) (WRONG CON) (RESP JES) (3 M ?)
       (2 U ?) (STIM PUM)

    7 TRUE IN PS
NOW INSERTING
(3 M ?) (1 P ?) => (USED) (DEP (REPLY JES)) (SAY JES)
ON LINE   5.3
    STM: (3 M ?) (RESP JES) (WRONG CON) (2 U ?)
       (STIM PUM)
```

Now memory contains a reply but no response, so rule 3 matches and elicits the correct response (JES) from the user. Rule 4 fires, since the reply differs from the response, marking the reply wrong. Next rule 5 fires, changing the USED label to a COND label. Finally rule 7 is reached and matches with the variables bound as: x1=3, x2=M, x3=JES, and x4=CON. When the rule is fired it now creates a new rule with <u>two</u> condition elements, one from the COND already in memory and one from the COND inserted by rule 7 itself. Note that this rule was inserted just above the previous rule that led to the error, thus insuring that in this ordered system it will be examined first. The two rules just added are:

```
(3 M ?) (1 P ?) => (USED) (DEP (REPLY JES)) (SAY JES)
        (1 P ?) => (USED (DEP (REPLY CON)) (SAY CON)
```

It should be clear that PAX will now elicit the response CON, and PUM the response JES, as desired.

The stimulus-response pairs given to EPAM1 for a test of paired-associate verbal learning are those of Figure 3. There were three training trials, and on the third trial the system made no errors. The output produced by the program is shown in Figure 5, and the trace of the test is shown in Appendix III.

The rules learned by the system are shown in Figure 6a in a shorthand notation.* These rules are equivalent to the discrimination net shown in Figure 6b. Note that the condition elements are analogous to intermediate nodes and the response elements to the terminal nodes in the net.

EPAM2. The second implementation of EPAM, called EPAM2, is shown in Figure 7. This more complete version of EPAM grows a production system similar to the one produced by EPAM1, except that the productions are analogous to a net in which response cues rather than complete responses are stored in some of the terminal nodes. These cues (i.e., C_N) are retrieved by dropping the stimulus through the net, and are then themselves dropped through the net to retrieve the responses stored in other terminal nodes. The system exhibits retroactive inhibition, stimulus generalization, and response generalization, as well as stimulus-response confusion (replying with a stimulus item instead of a response item).

Appendix IV gives a trace of EPAM2 learning the stimulus-response pairs of Figure 3. The output produced during the three training trials is shown in Figure 8. There were two instances of stimulus generalization, two of response generalization, one of both stimulus and response generalization, and two of stimulus-response confusion.

The production rules learned by EPAM2 and the corresponding discrimination net are shown in shorthand notation in Figures 9 and 10 respectively. Note

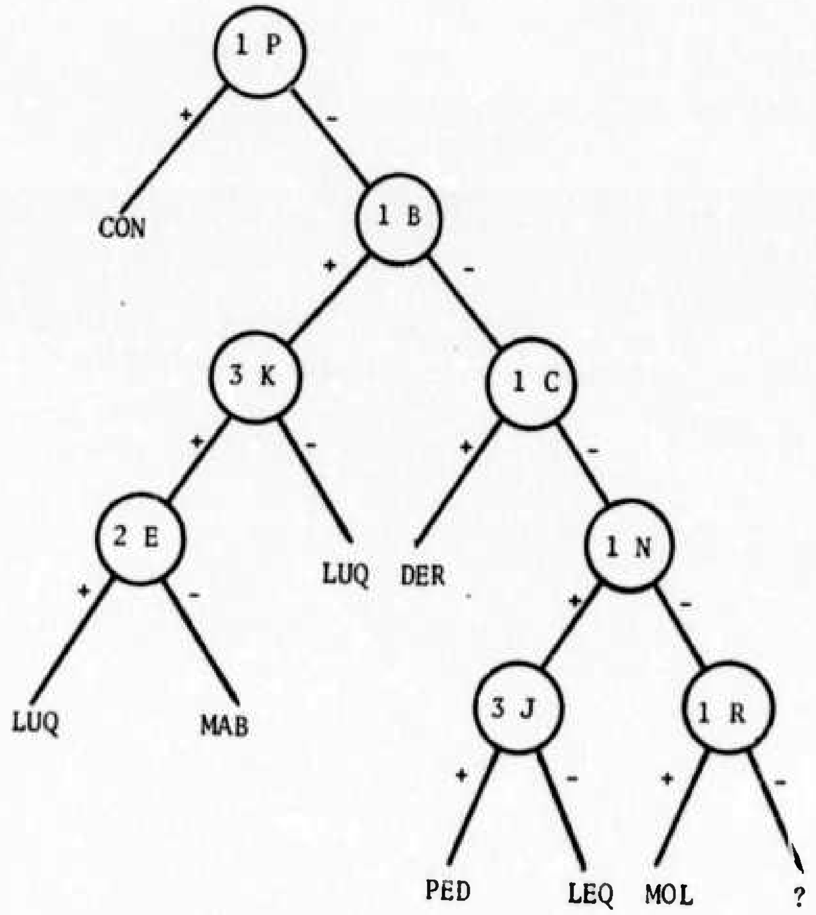
*Conditions, like (1 P), are elements denoting a letter and its location in the syllable, and are ordered (first, third, second) according to syllable location. Actions are response words like CON, or partial response cues like (1 M).

STIMULUS	REPLY 1	REPLY 2	REPLY 3	RESPONSE
PAX	?	CON	CON	CON
BEK	?	MAB (SG)	LUQ	LUQ
CIT	?	DER	DER	DER
BUK	LUQ (SG)	MAB	MAB	MAB
NAL	?	LEQ	LEQ	LEQ
REB	?	MOL	MOL	MOL
NOJ	LEQ (SG)	PED	PED	PED

Figure 5. EPAM1 Output for three Training Trials (SG indicates a stimulus generalization error)

```
          (1 P) => CON
(1 B)(3 K)(2 E) => LUQ
     (1 B)(3 K) => MAB
          (1 B) => LUQ
          (1 C) => DER
     (1 N)(3 J) => PED
          (1 N) => LEQ
          (1 R) => MOL
```

a. Production Rules Learned by EPAM1 (in shorthand notation)



b. Equivalent Discrimination Net

Figure 6. Production Rules Learned by EPAM1

```
1. (READY) (STIM X1) => (REM (READY)) (PERCEIVE X1 ?)
2. (READY) => (ATTEND STIM)
3. (REPLY) - (RESP) => (ATTEND RESP)
4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
5. (REPLY X1) (RESP X1) => (STOP)
6. (USED) (TEST X1) - (TEST X2) => (REP USED USED*)
7. (TEST X1) (TEST X2) (X3 X4 ?) => (REM (X3 X4 ?))
8. (TEST X1) (TEST X2) - (R-GEN) => (DEP (REPLY X1) (R-GEN)) (SAY X1)
9. - (RESP) => (DEP (REPLY ?)) (SAY ?) (ATTEND RESP)
10. (RESP X1) - (X2 X3 RESP) => (PERCEIVE X1 RESP)
11. (WRONG) (TEST X1) (STIM X1) - (R-GEN) => (DEP (R-GEN))
12. (OLD X1) (R-GEN) => (REP OLD COND) (DEP (HOLD X1))
13. (USED X1) (USED*) (R-GEN) => (REP USED COND) (DEP (HOLD X1))
14. (R-GEN) (COND (X1 X2 ?)) (X1 X2 RESP) => (REM (X1 X2 RESP))
15. (X1 X2 RESP) (RESP X3) (WRONG X4) - (DONE) => (COND (X1 X2 ?))
    (ACTION (OLD) (DEP (REPLY X3)) (SAY X3)) (PROD (SAY X4) (TEST X4)) (DEP (DONE))
16. (USED* X1) => (REP USED* COND)
17. (OLD) (DONE) - (TEST) => (REP OLD COND)
18. (R-GEN) (HOLD (X1 X2 ?)) => (REM (HOLD (X1 X2 ?))) (ACTION (DEP (X1 X2 ?)))
19. (R-GEN) (X1 X2 RESP) (STIM X3) (WRONG X4) => (ACTION (DEP (TEST X3)))
    (ACTION (USED) (DEP (X1 X2 ?))) (PROD (DEP (TEST X3))) (STOP)
20. (X1 X2 ?) (X3 X4 RESP) (STIM X5) (WRONG X6) => (COND (X1 X2 ?))
    (ACTION (USED) (DEP (X3 X4 ?)) (DEP (TEST X5))) (PROD (SAY X6)) (STOP)
```

Figure 7. EPAM2: A Production System Implementation of EPAM

STIMULUS	REPLY 1	REPLY 2	REPLY 3	RESPONSE
PAX	?	CON	CON	CON
BEK	?	MAB (SG)	LUQ	LUQ
CIT	CON (SR)	DER	DER	DER
BUK	LUQ (SG)	MAB	MAB	MAB
NAL	?	LUQ (RG)	LEQ	LEQ
REB	?	MAB (RG)	MOL	MOL
NOJ	LUQ (SG RG)	PAX (SR)	PED	PED

Figure 8. EPAM2 Output for Three Training Trials
(SG: stimulus generalization error,
RG: response generalization error,
SR: stimulus-response confusion).

```
          (1 D) => DER
     (1 C)(3 T) => (1 D)
          (1 C) => CON
     (1 P)(3 D) => PED
          (1 P) => (1 C)
     (1 L)(3 Q) => LEQ
          (1 L) => LUQ
(1 B)(3 K)(2 E) => (1 L)
     (1 M)(3 L) => MOL
          (1 M) => MAB
     (1 B)(3 K) => (1 M)
          (1 P) => PED
     (1 N)(3 J) => (1 P)(3 D)
     (1 N)(3 J) => (1 P)
          (1 L) => LUQ
          (1 B) => (1 L)
          (1 L) => LEQ
          (1 N) => (1 L)(3 Q)
          (1 N) => (1 L)
          (1 M) => MOL
          (1 R) => (1 M)(3 L)
          (1 R) => (1 M)
```

Figure 9. Production Rules Learned by EPAM2

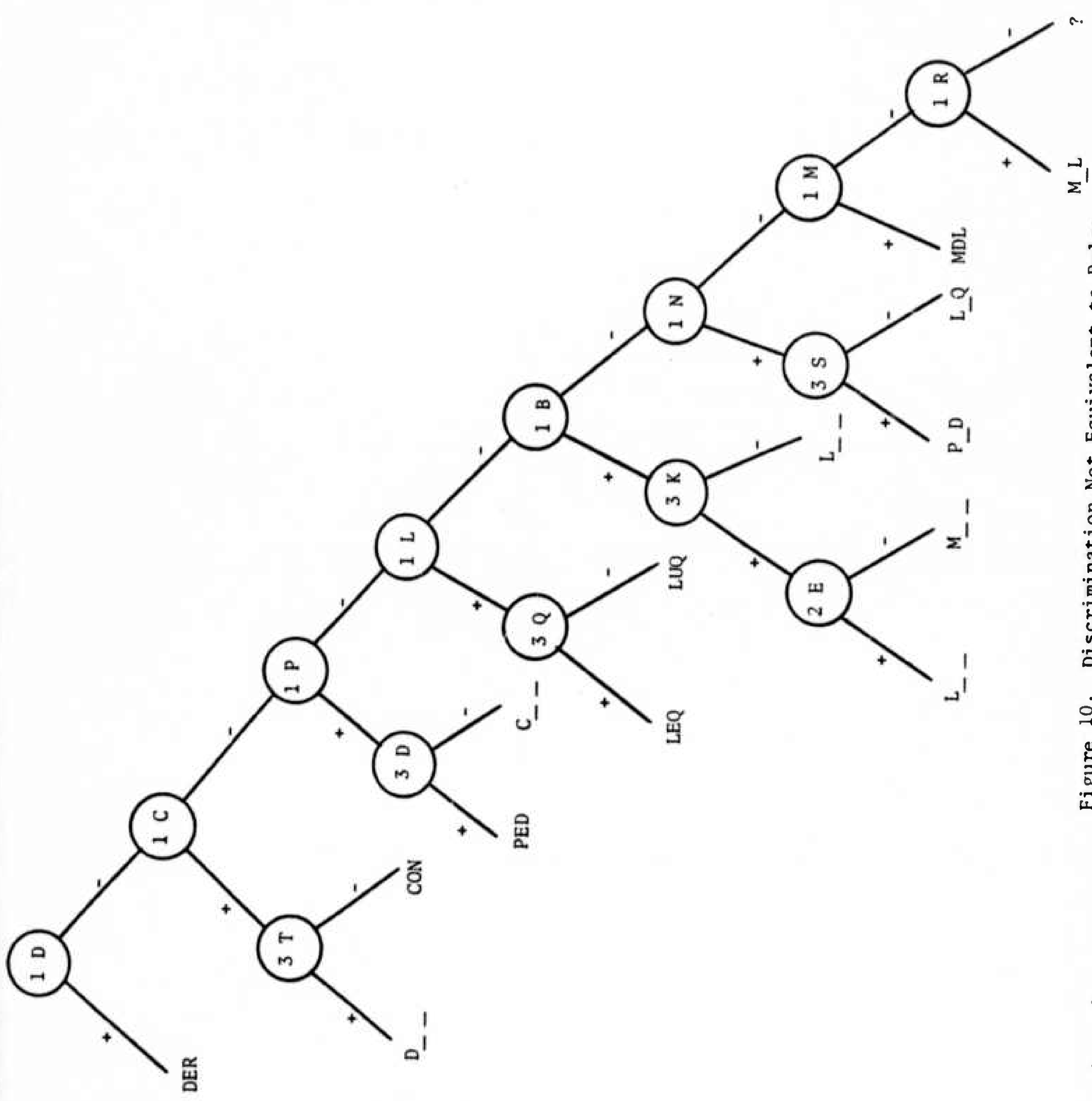


Figure 10. Discrimination Net Equivalent to Rules Learned by EPAM2 (Figure 9).

that the path through the net from the top node to a terminal node corresponds directly to the sequence of conditions tested in the production system to obtain a response.

EPAM2, illustrated in Figure 7, is an extremely compact piece of code which performs a sizable amount of information processing. Its power comes from the strong pattern matching capabilities inherent in the production system interpreter and from the primitive but highly useful memory modification and system building actions employed. Its compactness is due, in part, to the use of ordered production rules, since much information concerning rule applicability is implicit in the location of the rules.*

V. PRODUCTION SYSTEMS FOR SERIES COMPLETION TASKS

The series completion task has long been considered a learning task, but a very complex one having little in common with more fundamental learning tasks such as learning to add or learning to associate pairs of syllables. Computer models of series completion (Simon and Kotovsky, 1963; Klahr and Wallace, 1970; Williams, 1972; Hunt and Poltrock, 1974) have been complex programs with structures quite dissimilar from the structures of more basic learning models, such as EPAM.** Here an attempt is made to provide a common structure for these learning tasks The essence of their commonality is (1) an ordered production system representation of what is learned, and (2), the technique of adding new production rules above the error-causing rules to correct the errors.

* Unordered production system implementations of EPAM (Rychener, 1975) tend to require twice as many rules as EPAM2.

** The Hunt and Poltrock model is described as a collection of production systems. However, these systems are non-adaptive and do not represent the newly learned concept in production system form.

Simple Series Completion

The first production systems to be described are designed to solve simple repetition series completion problems, i.e., problems which can be solved using only the _same_ relation. Thus problems involving alphabets are excluded since they generally require the use of a _next_ or _prior_ relation on the alphabet. The extension of this approach to problems requiring other relations will be discussed later.

Basic Learning Technique. The learning technique used by the series completion production system is quite simple. Production rules are created such that the total set of rules represents a hypothesis about what symbols come next given a current context of symbols. The hypothesis is tested by checking the given series to see if the hypothesis (i.e., the current rules) correctly predicts each symbol in the series given the partial series up to that symbol. When every symbol in the problem series is correctly predicted, the system uses the hypothesis and the entire problem series to predict the next symbol in the series.

An example will illustrate the details of this procedure. Note that for this example the letters are considered unique, unordered symbols, that is they have no referent in an underlying alphabet. Consider the letter series ACABAC. First the series is partitioned as shown:

$$A \;/\; C\; A\; B\; A\; C\;. \qquad (4)$$

The current context A is matched against the current hypothesis to obtain a prediction for the next letter. At this point there are no hypotheses, except the default one which always says the next letter is the first letter in the series (in this case A). Since the prediction is incorrect* (C is

* The predictions made by the default hypothesis are always considered incorrect.

actually next, not A) a production rule is added above the existing rules. The condition side is the first letter of the current context (the first letter going right to left from the vertical line in 4). The action side is the letter immediately following the current context. Now there are two rules which constitute the hypothesis:

$$A \rightarrow C$$
$$xl \rightarrow A,$$

which say in effect, "if the first letter in the current context is A then predict C, otherwise predict A." Now the series is repartitioned as:

$$A\ C\ /\ A\ B\ A\ C \qquad (5)$$

and again the current context, this time A C, is matched against the hypothesis. Since the default rule again makes the prediction it is considered an error and a new rule is added as before to produce:

$$C \rightarrow A$$
$$A \rightarrow C$$
$$xl \rightarrow A \quad .$$

Now the series is repartitioned as:

$$A\ C\ A\ /\ B\ A\ C \qquad (6)$$

and the second rule incorrectly predicts that the next letter is C rather than B. A new rule is added, with just enough of the current context to distinguish it from the context in the rule that led to the error. Thus the rules become*:

$$C\ A \rightarrow B$$
$$C \rightarrow A$$
$$A \rightarrow C$$
$$xl \rightarrow A \quad .$$

*Notice that the new rule is added above all existing rules rather than above the error-causing rule. For simple series completion tasks this simplification will suffice.

The series is repartitioned as:

$$A\ C\ A\ B\ /\ A\ C \qquad (7)$$

which leads to a prediction by default rule and thus the addition of the rule B → A. The final partitioning leads to the correct prediction C, thus no new rule is added. The rules are now:

B → A

C A → B

C → A

A → C

x1 → A .

Since the end of the series has been reached and errors were made the partition-prediction process starts over with the series again partitioned as in 4-7. This time every prediction is correct so the learning phase (the phase in which new rules are added) is terminated. Now the entire series ACABAC is used as the context to predict the next letter. In this case the prediction is A. To extend the series the context plus predictions are used as context, thus ACABACA leads to a prediction of B. The extended series predicted by these rules is ACABAC $\underline{A}$ $\underline{B}$ $\underline{A}$ $\underline{C}$ $\underline{A}$ $\underline{B}$ ... .

<u>Letter Series</u>. The production system for simple letter series completion tasks is shown in Figure 11. The actions used are the same used in the previous systems except for OBSERVE (rather than PERCEIVE), and the addition of SUCC, the successor function on digits. The special-purpose OBSERVE action is defined below.

(OBSERVE $\underline{a}$ $\underline{b}$): Breaks word $\underline{a}$ into individual letters and tags these letters with $\underline{b}$.

```
 1. (READY) (SERIES X1) => (REP READY CONT)
    (OBSERVE X1 ?)
 2. (READY) => (ATTEND SERIES)
 3. (X1 ?) - (LOC) => (COND (1 X4 ?))
    (ACTION (DEP (NEXT X1))) (PROD END)
    (DEP (LOC X1))
 4. (0 X1 ?) => (SUCC)
 5. (ERROR) (SERIES X1) (LOC X2) - (X3 ?) =>
    (CLEAR (SERIES X1) (LOC X2)) (DEP (READY))
 6. (NEXT X1) - (X2 ?) => (SAY X1)
    (DEP (MATCH) (X1 ?)) (STOP)
 7. (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
    - (MATCH) => (DEP (MATCH))
 8. (USED) - (MATCH) - (ERROR) => (DEP (ERROR))
 9. (USED (X1 X2 ?)) (NEXT) => (REP USED OLD)
10. (X1 X2 ?) (NEXT) - (DONE) =>
    (REP (X1 X2 ?) (OLD (X1 X2 ?))) (DEP (DONE))
11. (OLD (X1 X2 ?)) => (REP OLD COND)
    (DEP (X1 X2 ?))
12. (NEXT X1) (MATCH) (SERIES X2) =>
    (REP (NEXT X1) CONT) (REM (MATCH) (DONE))
    (PROD (SERIES X2))
13. (LOC X1) (NEXT X2)
    (ACTION (USED) (DEP (NEXT X3))) => (REP X1 X3)
    (REP (NEXT X2) CONT 2) (REM (DONE))
    (PROD (NEXT X1))
14. (X1 ?) (CONT) (X2 ?) => (REP X1 (0 X1))
    (REM (CONT)) (ACTION (USED) (DEP (NEXT X2)))
15. (X1 ?) (CONT) => (REP X1 (0 X1)) (REM (CONT))
```

Figure 11. Production System for Simple Series Completion Task.

OBSERVE was designed for the series completion production systems and takes a letter series like ABABA and notices individual letters. For example, (OBSERVE ABABA ?) puts (A ?) (B ?) (A ?) (B ?) (A ?) into memory so that individual letters can be recognized. The OBSERVE and PERCEIVE actions are used rather than a set of more primitive actions because the primary interest here is not how words are perceived but rather how new information processing rules can be added to an existing system.

The first three rules of Figure 11 are initialization rules: they cause the series to be read in and the default rule to be created. Rule 13 fires when an erroneous prediction is made, adding a new rule to the system. Rule 12 fires when a correct prediction is made, removing all COND's and ACTION's from memory without adding a new rule*.

A trace of the system solving the series ABAB is shown in Appendix V. The rules learned for this series were:

$$B \rightarrow A$$
$$A \rightarrow B$$

Appendix VI shows a trace of the system solving the more complex series ABAACAABA. The rules learned are:

$$C\ A\ A \rightarrow B$$
$$C\ A \rightarrow A$$
$$C \rightarrow A$$
$$A\ A \rightarrow C$$

*Here PROD does not add a rule because it cannot find any current rule that contains the argument, i.e., (SERIES...).

$$B\ A \rightarrow A$$

$$B \rightarrow A$$

$$A \rightarrow B$$

which predict the extension of the series to be:

ABAACAABA $\underline{A}$ $\underline{C}$ $\underline{A}$ . . . .

Notice that the rule A → B is conditionally redundant*, that is, for this particular series it cannot be accessed. Figure 12 shows a number of simple letter series together with the rules learned by the letter series production system. Each set of rules represents the concept of the series, and the predictions made by them are similar to those obtained using the template model of Klahr and Wallace (1970).

Complex Letter Series Completion

A production system will now be described which can solve complex letter series completion tasks, i.e., letter series that may require the use of successor or predecessor operations on the alphabet.

Extended Learning Technique. The learning technique used here is similar to the one just described for simple series completion, in that production rules are created which represent hypotheses about which symbols come next given a current context of symbols. The major difference is that rules are generalized before they are added to the system. For example, in the simple series completion program with context C A and next letter B the specific rule C A → B is added. But here a generalized version of this rule is added which takes into account the letter relationships which might be relevant. The problem is that the rules can be generalized in a number of different ways, each way being a hypothesis about which letter relationships are relevant for this

*See Waterman (1970) for a discussion of redundancy in ordered production systems.

Series	Rules	Prediction
1. ABAABAAB	B A → A	AAB
	B → A	
	A → B	
2. AAABAAABAA	B A A → A	ABA
	B → A	
	A A → B	
	A → A	
3. CAABACAAB	B A → C	ACA
	B → A	
	A A → B	
	A → A	
	C → A	
4. ABABCABAB	B A B → C	CAB
	A B → A	
	C → A	
	A → B	

Figure 12. Rules Learned by Simple Letter Series Completion Production System (Redundant rules not shown).

particular series. The variations on C A → B are shown below.

x1 A → B

C x1 → B

x1 A → 'x1

C x1→ x1'

x1 x2 → B

x1 ''x1 → B

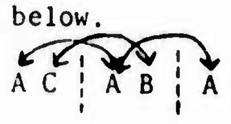
x1 x2 → 'x1

x1 ''x1 → 'x1

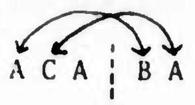
The first rule above is interpreted "any letter followed by A leads to B", the second is "C followed by any letter leads to B", the third "any letter followed by A leads to the predecessor of that letter", etc.

If every time a new rule was added, the system arbitrarily picked a generalization, intending to backtrack to try the other generalizations when an error was discovered, a huge tree of possibilities would be generated, making the problem virtually unsolvable. The solution to this dilemma is to use tree-pruning heuristics to limit the number of possible generalizations at each step. The production system to be described uses one very powerful heuristic which will be called the <u>template heuristic</u>.

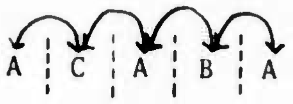
The template heuristic consists of hypothesizing the period size, and recognizing only relations between letters which occupy the same relative position within the period, while generalizing on all letters. For example, if given the series ACABA with assumed period 2, then the relations looked for are shown by the arrows below.



For context CA and next letter B only one generalized rule, x1 x2 → 'x1, can be obtained. For the same series with assumed period 3 the relations looked for are:



and only the rule x1 x2 → B can be obtained. For assumed period 1 the relations are:



and only x1 ''x1 → 'x1 can be obtained. These rules are all examples of inter-period rules, since in each case the context and next letter are not all located within the same period.

Predecessor and successor relations between letters are always represented by letting the variable stand for the first letter and the variable plus appropriate apostrophes stand for the second letter, i.e., A and C would be x1 and x1'', while C and A would be x1 and ''x1. The <u>same</u> relation can be represented in either of two ways. The first, total generalization, involves substituting the same variable for each letter. The second, partial generalization, consists of using the letter itself rather than a variable. Thus if given the context A B A C the relation between alternate letters could be represented as x1 x2 x1 x2' (total generalization) or as A x2 A x2' (partial generalization). The system will generate correct concepts regardless of whether it uses total or partial generalization on <u>same</u>. But concepts for simple repetition series tend to be more concise when total generalization is used, consequently, a combination of both is used in the actual implementation.*

*The first inter-period rule added during each new period size hypothesis uses total generalization on <u>same</u>. The rest use partial generalization.

Complex series completion learning proceeds as follows: period size is hypothesized and the series goes through a partition-prediction cycle as in the simple series completion technique. The differences are that generalized rules are added, and the partition-prediction process is not repeated until every prediction is correct; it is performed just once for each period hypothesis. A period hypothesis is considered false if:

(1) no relation can be established between letters which occupy the same relative position within the period,

or (2) the number of inter-period* rules added exceeds the current period size hypothesis.

When the current period size hypothesis is found to be false, a period size one greater than the previous is hypothesized, and the entire partition-prediction process starts over from the beginning. An example will clarify this procedure. Consider the series ABHBCICD. The initial period size hypothesis is 1 and the initial partitioning is shown below.

A /¦B¦H¦B¦C¦I¦C¦D

The system already contains the default rule $x1 \rightarrow x1$, which is always considered to generate an error. Thus the context A is dropped through the rules and the erroneous prediction A is made. Now the system takes the context A and next letter B to form $A \rightarrow B$, generalizes it to get $x1 \rightarrow x1'$, and places it above the error-causing (default) rule to produce:

$$x1 \rightarrow x1' \qquad (1)$$

*Intra-period rules are not counted (since they are not needed to extend the series) and are always considered to lead to an error.

The next partitioning is

A¦B /¦H¦B¦C¦I¦C¦D

which produces the context AB and next letter H. The context matches rule (1) but the prediction is C. Thus the system adds a new rule, a generalized version of A B → H. But this makes the number of rules added (2) exceed the period hypothesis (1) so a new hypothesis of 2 is generated. Partitioning starts over as:

A / B¦H B¦C I¦C D .

and the rules again become just the default rule x1 → x1. The context A leads to a prediction of A (from the default rule) and an intra-period rule, the generalization of A → B, is added. The rules are now:

x1 → B (initialization).

The intra-period rules are called initialization rules because they are not needed to extend the series, only to generate it from scratch. The next partitioning is

A B /¦H B¦C I¦C D ,

and the context B matches the first rule, leading to the erroneous prediction B. Now the rule A B → H is generalized, but since no relation between A and H can be found* the period 2 hypothesis fails. Partitioning starts over as:

A / B H¦B C I¦C D

and the rules again become just the default rule. Context A generates a prediction of A which leads to the intra-period rule A → B being

*The system does not search for relations higher than triple predecessor or successor.

generalized and added to produce:

$x1 \rightarrow B$ (initialization).

The next partitioning is

A B / H ¦ B C I ¦ C D

and context B matches the top rule, leading to the erroneous prediction B. Now the intra-period rule $A\ B \rightarrow H$ is generalized and added to give:

$x1\ x2 \rightarrow H$ (initialization)

$x1 \rightarrow B$ (initialization).

The next partitioning is

A B H / ¦ B C I ¦ C D

and context BH matches the top rule leading to the prediction H. Now the inter-period rule $A\ B\ H \rightarrow B$ is generalized and added above the error-causing rule to give:

$x1\ x2\ x3 \rightarrow x1'$ (1)

$x1\ x2 \rightarrow H$ (initialization)

$x1 \rightarrow B$ (initialization).

The next partitioning

A B H ¦ B / C I ¦ C D

produces the context A B H B, and the B H B matches the x1 x2 x3 of rule (1) to produce the correct prediction C. Since the prediction was correct no rules are added and the partitioning continues 3 more times, each producing the correct prediction. The partition-prediction cycle is now complete, and the entire series is taken as context to produce the next letter prediction. Rule (1) fires and predicts J. The concept of the series is now embodied in the numbered rules (the inter-period rules). Thus we say that $x1\ x2\ x3 \rightarrow x1'$ is the concept learned by the system, and the series predicted by this concept is ABHBCICD<u>JDEK</u> ... .

Production System. The production system for complex letter series completion is shown in Figure 13. The template heuristic is embodied in the special action (PRODS a b c d). The argument b is the hypothesized period (initially 1), and c is the series itself. As in PROD, memory elements marked (COND ...) and (ACTION ...) are combined and removed from memory to create a new rule. This rule is generalized according to the template heuristic and placed immediately above the first existing rule that contains the argument a. The argument d is the number of rules already added to the production system by PRODS. The action PRODS is clearly non primitive and could have been written as a separate production system which used the action PROD. It was written as an action rather than a production system so as not to obscure the production modification techniques being illustrated.

Rules 1 and 2 in Figure 13 provide initialization, rule 16 is the default rule, and rule 13 adds productions to the system. Appendix VII shows the trace of this production system on the series CDCDCD and ABMCDMEF. The concept learned for CDCDCD (in shorthand notation) is:

$$x1\ x2 \rightarrow x1\ .$$

Note that this is the concept of any simple repetition series of period 2, not simply the concept of the series CDCDCD. In general, any simple repetition series of length n, will be learned as x1 x2 x3 ... xn → x1.

Figure 14 shows concepts learned using the 15 series from Simon & Kotovsky (1963). The correct predictions are made in all cases, but in series 8 the system generated the concept of a series with period 6 rather than the simpler concept of a series with period 3. In general, this learning technique will successfully solve letter series completion problems that can be solved by Klahr's template model (Klahr & Wallace, 1970), if given enough of the series. However, this particular algorithm does not guarantee finding the concept of the shortest period, and may instead, as in series 8, find multiples of the shortest period. For more on serial pattern acquisition see Waterman (1975).

```
 1. (READY) (SERIES X1)  =>  (REP READY CONT)
    (DEP (PNUM 2) (COUNTS 0)) (OBSERVES X1 ?)
 2. (READY)  =>  (ATTEND SERIES) (DEP (PERIOD 1))
 3. (COUNT) (COUNTS X1)  =>  (REM (COUNT))
    (REP X1 X1')
 4. (0 X1 ?)  =>  (SUCC)
 5. (FAIL) (PERIOD X1) (SERIES X2)  =>  (ERASE)
    (CLEAR) (DEP (READY) (PERIOD X1') (SERIES X2))
 6. (PERIOD X1) (COUNTS X1') (SERIES X2)  =>
    (ERASE) (CLEAR)
    (DEP (READY) (PERIOD X1') (SERIES X2))
 7. (NEXT X1) - (X2 ?) - (ACTION)  =>  (SAY X1)
    (DEP (MATCH) (X1 ?)) (STOP)
 8. (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
    - (MATCH) - (ERROR)  =>  (DEP (MATCH))
 9. (X1 X2 ?) (NEXT) - (DONE)  =>
    (DEP (OLD (X1 X2 ?))) (DEP (DONE))
10. (USED (X1 X2 ?))  =>  (REP USED OLD)
    (DEP (X1 X2 ?))
11. (OLD (X1 X2 ?))  =>  (REP OLD COND)
12. (MATCH) (NEXT X1) (SERIES X2) (LOC X3)  =>
    (REP (NEXT X1) CONT 2)
    (REM (MATCH) (DONE) (LOC X3)) (PROD (SERIES X2))
13. (LOC X1) (NEXT X2) (PERIOD X3) (SERIES X4)
    (COUNTS X5)  =>  (REM (LOC X1) (DONE) (ERROR))
    (REP (NEXT X2) CONT) (PRODS (LOC X1) X3 X4 X5)
14. (CONT) (X1 ?) (PNUM X2) (X3 ?)  =>
    (REP X1 (0 X1) 2) (REP X2 X2' 3) (REM (CONT))
    (ACTION (USED) (DEP (NEXT X3)) (DEP (LOC X2)))
15. (CONT) (X1 ?)  =>  (REP X1 (0 X1) 2)
    (REM (CONT))
16. (1 X1 ?)  =>  (DEP (NEXT X1) (LOC 1))
```

Figure 13. Production System for Complex Series Completion Task.

	Series	Rules	Prediction
1.	CDCDCD	x1 x2 → x1	CDC
2.	AAABBB	x1 x2 x3 → x1'	CCC
3.	ATBATAATB	x1 x2 x3 x4 x5 x6 → x1	ATA
4.	ABMCDMEF	x1 M x3 x1'' → M x1 x2 x3 → x1''	MGH
5.	DEFGEFGH	x1 x2 x3 x4 → x1'	FGH
6.	QXAPXBQXA	x1 x2 x3 x4 x5 x6 → x1	PXB
7.	ADUACUAEUABUAF	x1 U A x4 x5 x6 'x1 U A → x4' U A x3 x4 x5 x6 U A → 'x3 A x2 x3 x4 x5 x6 A → x2' x1 U x3 x4 x5 x6 'x1 → U x1 x2 x3 x4 x5 x6 → x1	UAA
8.	MABMBCMCDM	M x2 x3 x4 x5 x6 M → x2'' x1 x2 M x4 x5 x6 x1'' x2'' → M x1 x2 x3 x4 x5 x6 x1'' → x2'' x1 x2 x3 x4 x5 x6 → x1	DEM
9.	URTUSTU	U x2 x3 U → x2' x1 x2 x3 → x1	TTU
10.	ABYABXAB	B x2 x3 B → 'x2 x1 x2 x3 → x1	WAB
11.	RSCDSTDE	x1 x2 x3 x4 → x1'	TUE
12.	NPAOQAPR	x1 A x3 x1' → A x1 x2 x3 → x1'	AQS
13.	WXAXYBY	x1 x2 x3 → x1'	ZCZ
14.	JKQRKLRS	x1 x2 x3 x4 → x1'	LMS
15.	PONONMNM	x1 x2 x3 → 'x1	LML

Figure 14. Rules Learned by Complex Series Completion Production System

VI. CONCLUSION

The PAS-II production system has been described and used to illustrate adaptive techniques in production system construction. The focus has been on the machinery needed to implement self-modification in a production system framework. It has been demonstrated that a production building action (PROD) is the crucial one needed for such an implementation, and that its use in an ordered production system leads to relatively short, straight-forward programs.

Moreover, it has been demonstrated that within a production system representation, using the actions illustrated, one can create a learning paradigm which applies to (1) very simple rote learning tasks such as learning the addition table, (2) more involved learning tasks like nonsense syllable association and discrimination, and (3) complex induction tasks such as inducing the concept of a serial pattern. In all three cases the paradigm consisted of creating an ordered production system representation of the concept learned by adding new production rules (or hypotheses) above the error-causing rules.

The adaptive production systems described in this paper are by nature quite parsimonious; that is, the system which learns the concept is represented in exactly the same fashion as the concept being learned. They are both represented as production rules in a single production system. This eliminates the need for having two different types of control in the system; one for activating the learning mechanism and another for accessing the concept learned. Another way of stating this is that the concepts learned are not passive, static structures which must be given a special interpretation, but rather are self-contained programs which are executed automatically in the course of executing the learning mechanism.

It should be stressed that much of the system simplicity seen in the production system examples is due to using ordered production rules with powerful pattern matching capabilities. With ordered rules the system can use the simple heuristic "add a new rule immediately above the one that made the error" to great advantage, as illustrated by the EPAM and series completion examples.

Finally, the analogy between an ordered production system and a discrimination net has been made clear, i.e., that the condition elements are non-terminal nodes in the net, the action elements are terminal nodes, and the searches through the conditions in the production system are analogous to the paths from the top element to the terminal elements in the net.

ACKNOWLEDGMENTS

The author wishes to thank David Klahr for his many ideas and suggestions concerning this paper and the work on which it is based. The comments and criticisms of Dick Hayes, Herbert Simon, and Allen Newell on the final draft of this paper are also gratefully acknowledged. This work has been supported in part by the NIH Grant MH-07722, and in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense, (1-58200-8130).

REFERENCES

Deese, J., and Kaufman, R. A. Serial effects in recall of unorganized and sequentially organized verbal material. Journal of Experimental Psychology, 1957, 54, 180-187.

Feigenbaum, E. A. The simulation of verbal learning behavior. In Feigenbaum, E., and Feldman, J. (Eds.), Computers and Thought. McGraw-Hill, New York, 1963, pp. 297-309.

Feigenbaum, E. A., and Simon, H. A. An information processing theory of some effects of similarity, familiarization, and meaningfulness in verbal learning. Journal of Verbal Learning and Verbal Behavior, Vol. 3, 1964, pp. 385-396.

Hunt, E. B., and Poltrock, S. E. The mechanics of thought. In Kantowitz, B. (Ed.), Human Information Processing: Tutorials in Performance and Cognition, Lawrence Erlbaum, N.J., 1974, pp. 277-350.

Klahr, D., and Wallace, J. G. The development of serial completion strategies: An information processing analysis. British Journal of Psychology, Vol. 61, 1970, pp. 243-257.

Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In Melton, A. W., and Marton, E. (Eds.), Coding Processes in Human Memory, Washington, D.C., Winston and Sons, 1972.

Newell, A. Production systems: Models of control structures. Visual Information Processing, Chase, W. (Ed.), Academic Press, 1973.

Newell, A., and Simon, H. A. Human Problem Solving. Englewood Cliffs, N.J., Prentice Hall, 1972.

Rychener, M. Production Systems as a Programming Language for AI Applications, Ph.D. Thesis, in preparation, Computer Science Department, CMU, 1975.

Simon, H. A., and Kotovsky, K. Human acquisition of concepts for sequential patterns. Psychological Review, Vol. 70, no. 6, 1963, pp. 534-546.

Waterman, D. A. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence, Vol. 1, no. 1 and 2, 1970, pp. 121-170.

Waterman, D. A. PAS-II Reference Manual. Computer Science Department Report, CMU, June, 1973.

Waterman, D. A. Serial pattern acquisition: A production system approach. CIP Working Paper #286, CMU, February, 1975.

Waterman, D. A., and Newell, A. PAS-II: An interactive task-free version of an automatic protocol analysis system. Proceedings of the Third IJCAI, 1973, pp. 431-445.

Williams, D. S. Computer program organization induced from problem examples. In Simon, H. A., and Siklossy, L. (Eds.), Representation and Meaning, Prentice Hall, Englewood Cliffs, N.J., 1972, pp. 143-205.

APPENDIX I. Trace of ADD Production System.

```
•memory display ps display
MEMORY MODE
     1. STM = (READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

PS MODE
     1. (READY) (ORDER X1) => (REP (READY) (COUNT X1))
        (ATTEND)
     2. (N X1) - (NN) - (S NN) => (DEP (NN X1))
     3. (COUNT X1) (M X1) (NN X2) (N X3) =>
        (SAY X2 IS THE ANSWER) (COND (M X1) (N X3))
        (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER))
        (PROD) (STOP)
     4. (COUNT) (NN) => (REP (COUNT) (S COUNT))
        (REP (NN) (S NN) 2)
     5. (ORDER X1 X2) => (REP (X1 X2) (X2))
        (COND (S X3 X1))
        (ACTION (REP (S X3 X1) (X3 X2))) (PROD)

•fire
     1 TRUE IN PS
OUTPUT FOR (ATTEND) = (dep (m 4)(n 2))
     STM: (N 2) (M 4) (COUNT 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     2 TRUE IN PS
     STM: (NN 2) (N 2) (M 4) (COUNT 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     4 TRUE IN PS
     STM: (S COUNT 0) (S NN 2) (N 2) (M 4)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     5 TRUE IN PS
NOW INSERTING
(S X3 0) => (REP (S X3 0) (X3 1))
ON LINE 0.5
     STM: (ORDER 1 2 3 4 5 6 7 8 9) (S COUNT 0) (S NN 2)
        (N 2) (M 4)

    0.5 TRUE IN PS
     STM: (COUNT 1) (ORDER 1 2 3 4 5 6 7 8 9) (S NN 2)
        (N 2) (M 4)

     5 TRUE IN PS
NOW INSERTING
(S X3 1) => (REP (S X3 1) (X3 2))
ON LINE 0.25
     STM: (ORDER 2 3 4 5 6 7 8 9) (COUNT 1) (S NN 2)
        (N 2) (M 4)

     5 TRUE IN PS
NOW INSERTING
(S X3 2) => (REP (S X3 2) (X3 3))
ON LINE 0.13
     STM: (ORDER 3 4 5 6 7 8 9) (COUNT 1) (S NN 2) (N 2)
        (M 4)

  0.13 TRUE IN PS
     STM: (NN 3) (ORDER 3 4 5 6 7 8 9) (COUNT 1) (N 2)
        (M 4)

     4 TRUE IN PS
     STM: (S COUNT 1) (S NN 3) (ORDER 3 4 5 6 7 8 9)
        (N 2) (M 4)

  0.25 TRUE IN PS
     STM: (COUNT 2) (S NN 3) (ORDER 3 4 5 6 7 8 9) (N 2)
        (M 4)

     5 TRUE IN PS
NOW INSERTING
(S X3 3) => (REP (S X3 3) (X3 4))
ON LINE 0.05
     STM: (ORDER 4 5 6 7 8 9) (COUNT 2) (S NN 3) (N 2)
        (M 4)

  0.05 TRUE IN PS
     STM: (NN 4) (ORDER 4 5 6 7 8 9) (COUNT 2) (N 2)
        (M 4)

     4 TRUE IN PS
     STM: (S COUNT 2) (S NN 4) (ORDER 4 5 6 7 8 9) (N 2)
        (M 4)

  0.13 TRUE IN PS
     STM: (COUNT 3) (S NN 4) (ORDER 4 5 6 7 8 9) (N 2)
        (M 4)

     5 TRUE IN PS
NOW INSERTING
(S X3 4) => (REP (S X3 4) (X3 5))
ON LINE 0.03
     STM: (ORDER 5 6 7 8 9) (COUNT 3) (S NN 4) (N 2)
        (M 4)

  0.03 TRUE IN PS
     STM: (NN 5) (ORDER 5 6 7 8 9) (COUNT 3) (N 2) (M 4)

     4 TRUE IN PS
     STM: (S COUNT 3) (S NN 5) (ORDER 5 6 7 8 9) (N 2)
        (M 4)

  0.05 TRUE IN PS
     STM: (COUNT 4) (S NN 5) (ORDER 5 6 7 8 9) (N 2)
        (M 4)

     5 TRUE IN PS
NOW INSERTING
(S X3 5) => (REP (S X3 5) (X3 6))
ON LINE 0.02
     STM: (ORDER 6 7 8 9) (COUNT 4) (S NN 5) (N 2) (M 4)

  0.02 TRUE IN PS
     STM: (NN 6) (ORDER 6 7 8 9) (COUNT 4) (N 2) (M 4)

     3 TRUE IN PS

6 IS THE ANSWER

NOW INSERTING
(M 4) (N 2) => (SAY 6 IS THE ANSWER) (STOP)
ON LINE 0.01
     STM: (COUNT 4) (M 4) (NN 6) (N 2) (ORDER 6 7 8 9)

•display
  0.01. (M 4) (N 2) => (SAY 6 IS THE ANSWER) (STOP)
  0.02. (S X3 5) => (REP (S X3 5) (X3 6))
  0.03. (S X3 4) => (REP (S X3 4) (X3 5))
  0.05. (S X3 3) => (REP (S X3 3) (X3 4))
  0.13. (S X3 2) => (REP (S X3 2) (X3 3))
```

```
 0.25  (S X3 1)  =>  (REP (S X3 1) (X3 2))
  0.5  (S X3 0)  =>  (REP (S X3 0) (X3 1))
    1. (READY) (ORDER X1)  =>  (REP (READY) (COUNT X1))
       (ATTEND)
    2. (N X1) - (NN) - (S NN)  =>  (DEP (NN X1))
    3. (COUNT X1) (M X1) (NN X2) (N X3)  =>
       (SAY X2 IS THE ANSWER) (COND (M X1) (N X3))
       (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER))
       (PROD) (STOP)
    4. (COUNT) (NN)  =>  (REP (COUNT) (S COUNT))
       (REP (NN) (S NN) 2)
    5. (ORDER X1 X2)  =>  (REP (X1 X2) (X2))
       (COND (S X3 X1))
       (ACTION (REP (S X3 X1) (X3 X2))) (PROD)

•initialize fire
INITIALIZED
     1 TRUE IN PS
OUTPUT FOR (ATTEND) = (dep (m 4)(n 2))
     STM: (N 2) (M 4) (COUNT 0)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

  0.01 TRUE IN PS

6 IS THE ANSWER

     STM: (M 4) (N 2) (COUNT 0)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

•initialize fire
INITIALIZED
     1 TRUE IN PS
OUTPUT FOR (ATTEND) = (dep (m 2)(n 1))
     STM: (N 1) (M 2) (COUNT 0)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

     2 TRUE IN PS
     STM: (NN 1) (N 1) (M 2) (COUNT 0)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

     4 TRUE IN PS
     STM: (S COUNT 0) (S NN 1) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

  0.25 TRUE IN PS
     STM: (NN 2) (S COUNT 0) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

   0.5 TRUE IN PS
     STM: (COUNT 1) (NN 2) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

     4 TRUE IN PS
     STM: (S COUNT 1) (S NN 2) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

  0.13 TRUE IN PS
     STM: (NN 3) (S COUNT 1) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

  0.25 TRUE IN PS
     STM: (COUNT 2) (NN 3) (N 1) (M 2)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

     3 TRUE IN PS

3 IS THE ANSWER

NOW INSERTING
(M 2) (N 1)  =>  (SAY 3 IS THE ANSWER) (STOP)
ON LINE 0.005
     STM: (COUNT 2) (M 2) (NN 3) (N 1)
          (ORDER 0 1 2 3 4 5 6 7 8 9)

•display
0.005  (M 2) (N 1)  =>  (SAY 3 IS THE ANSWER) (STOP)
 0.01  (M 4) (N 2)  =>  (SAY 6 IS THE ANSWER) (STOP)
 0.02  (S X3 5)  =>  (REP (S X3 5) (X3 6))
 0.03  (S X3 4)  =>  (REP (S X3 4) (X3 5))
 0.05  (S X3 3)  =>  (REP (S X3 3) (X3 4))
 0.13  (S X3 2)  =>  (REP (S X3 2) (X3 3))
 0.25  (S X3 1)  =>  (REP (S X3 1) (X3 2))
  0.5  (S X3 0)  =>  (REP (S X3 0) (X3 1))
    1. (READY) (ORDER X1)  =>  (REP (READY) (COUNT X1))
       (ATTEND)
    2. (N X1) - (NN) - (S NN)  =>  (DEP (NN X1))
    3. (COUNT X1) (M X1) (NN X2) (N X3)  =>
       (SAY X2 IS THE ANSWER) (COND (M X1) (N X3))
       (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER))
       (PROD) (STOP)
    4. (COUNT) (NN)  =>  (REP (COUNT) (S COUNT))
       (REP (NN) (S NN) 2)
    5. (ORDER X1 X2)  =>  (REP (X1 X2) (X2))
       (COND (S X3 X1))
       (ACTION (REP (S X3 X1) (X3 X2))) (PROD)
```

```
•memory display ps display
MEMORY MODE
    1. STM = (READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

PS MODE
    1. (READY) => (REP (READY) (COUNT 0))
       (DEP (ANS 0)) (ATTEND)
    2. (M X1) - (REMAIN) => (DEP (REMAIN X1))
    3. (ANS) (COUNT X1) (REMAIN X3) (N X1) (M X2)
        => (REP (ANS) (S ANS)) (REP (X1) (0) 2)
       (REP (X3) (X2) 3)
    4. (M 0) (ANS X1) (REMAIN X2) - (S) =>
       (SAY X1 WITH REMAINDER X2) (STOP)
    5. (M) (COUNT) - (S) => (REP (M) (P M))
       (REP (COUNT) (S COUNT) 2)
    6. (ORDER X1 X2) - (GO) => (REP (X1 X2) (X2))
       (COND (P X3 X2))
       (ACTION (REP (P X3 X2) (X3 X1))) (PROD (GO))
       (COND (S X3 X1))
       (ACTION (REP (S X3 X1) (X3 X2))) (PROD (READY))

•fire
    1 TRUE IN PS
OUTPUT FOR (ATTEND) = (dep (m 3)(n 2))
    STM (N 2) (M 3) (ANS 0) (COUNT 0)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    2 TRUE IN PS
    STM (REMAIN 3) (M 3) (N 2) (ANS 0) (COUNT 0)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    5 TRUE IN PS
    STM (P M 3) (S COUNT 0) (REMAIN 3) (N 2) (ANS 0)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    6 TRUE IN PS
NOW INSERTING
(P X3 1) => (REP (P X3 1) (X3 0))
ON LINE  5.5
NOW INSERTING
(S X3 0) => (REP (S X3 0) (X3 1))
ON LINE  0.5
    STM (ORDER 1 2 3 4 5 6 7 8 9) (P M 3) (S COUNT 0)
       (REMAIN 3) (N 2) (ANS 0)

  0.5 TRUE IN PS
    STM (COUNT 1) (ORDER 1 2 3 4 5 6 7 8 9) (P M 3)
       (REMAIN 3) (N 2) (ANS 0)

    6 TRUE IN PS
NOW INSERTING
(P X3 2) => (REP (P X3 2) (X3 1))
ON LINE  5.8
NOW INSERTING
(S X3 1) => (REP (S X3 1) (X3 2))
ON LINE  0.8
    STM (ORDER 2 3 4 5 6 7 8 9) (COUNT 1) (P M 3)
       (REMAIN 3) (N 2) (ANS 0)

    6 TRUE IN PS
NOW INSERTING
(P X3 3) => (REP (P X3 3) (X3 2))
ON LINE  5.9
NOW INSERTING
(S X3 2) => (REP (S X3 2) (X3 3))
ON LINE  0.9
    STM (ORDER 3 4 5 6 7 8 9) (COUNT 1) (P M 3)
       (REMAIN 3) (N 2) (ANS 0)

  5.9 TRUE IN PS
    STM (M 2) (ORDER 3 4 5 6 7 8 9) (COUNT 1)
       (REMAIN 3) (N 2) (ANS 0)

    5 TRUE IN PS
    STM (P M 2) (S COUNT 1) (ORDER 3 4 5 6 7 8 9)
       (REMAIN 3) (N 2) (ANS 0)

  0.8 TRUE IN PS
    STM (COUNT 2) (P M 2) (ORDER 3 4 5 6 7 8 9)
       (REMAIN 3) (N 2) (ANS 0)

  5.8 TRUE IN PS
    STM (M 1) (COUNT 2) (ORDER 3 4 5 6 7 8 9)
       (REMAIN 3) (N 2) (ANS 0)

    3 TRUE IN PS
    STM (S ANS 0) (COUNT 0) (REMAIN 1) (N 2) (M 1)
       (ORDER 3 4 5 6 7 8 9)

  0.5 TRUE IN PS
    STM (ANS 1) (COUNT 0) (REMAIN 1) (N 2) (M 1)
       (ORDER 3 4 5 6 7 8 9)

    5 TRUE IN PS
    STM (P M 1) (S COUNT 0) (ANS 1) (REMAIN 1) (N 2)
       (ORDER 3 4 5 6 7 8 9)

  0.5 TRUE IN PS
    STM (COUNT 1) (P M 1) (ANS 1) (REMAIN 1) (N 2)
       (ORDER 3 4 5 6 7 8 9)

  5.5 TRUE IN PS
    STM (M 0) (COUNT 1) (ANS 1) (REMAIN 1) (N 2)
       (ORDER 3 4 5 6 7 8 9)

    4 TRUE IN PS

1 WITH REMAINDER 1

    STM (M 0) (ANS 1) (REMAIN 1) (COUNT 1) (N 2)
       (ORDER 3 4 5 6 7 8 9)

•display
  0.5 (S X3 0) => (REP (S X3 0) (X3 1))
  0.8 (S X3 1) => (REP (S X3 1) (X3 2))
  0.9 (S X3 2) => (REP (S X3 2) (X3 3))
    1. (READY) => (REP (READY) (COUNT 0))
       (DEP (ANS 0)) (ATTEND)
    2. (M X1) - (REMAIN) => (DEP (REMAIN X1))
    3. (ANS) (COUNT X1) (REMAIN X3) (N X1) (M X2)
        => (REP (ANS) (S ANS)) (REP (X1) (0) 2)
       (REP (X3) (X2) 3)
    4. (M 0) (ANS X1) (REMAIN X2) - (S) =>
       (SAY X1 WITH REMAINDER X2) (STOP)
    5. (M) (COUNT) - (S) => (REP (M) (P M))
       (REP (COUNT) (S COUNT) 2)
  5.5 (P X3 1) => (REP (P X3 1) (X3 0))
  5.8 (P X3 2) => (REP (P X3 2) (X3 1))
  5.9 (P X3 3) => (REP (P X3 3) (X3 2))
    6. (ORDER X1 X2) - (GO) => (REP (X1 X2) (X2))
```

```
        (COND (P X3 X2))
        (ACTION (REP (P X3 X2) (X3 X1))) (PROD (GO))
        (COND (S X3 X1))
        (ACTION (REP (S X3 X1) (X3 X2))) (PROD (READY))

 •initialize fire
 INITIALIZED
     1 TRUE IN PS
 OUTPUT FOR (ATTEND) = (dep (m 2)(n 1))
     STM (N 1) (M 2) (ANS 0) (COUNT 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     2 TRUE IN PS
     STM (REMAIN 2) (M 2) (N 1) (ANS 0) (COUNT 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     5 TRUE IN PS
     STM (P M 2) (S COUNT 0) (REMAIN 2) (N 1) (ANS 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    0.5 TRUE IN PS
     STM (COUNT 1) (P M 2) (REMAIN 2) (N 1) (ANS 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    5.8 TRUE IN PS
     STM (M 1) (COUNT 1) (REMAIN 2) (N 1) (ANS 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     3 TRUE IN PS
     STM (S ANS 0) (COUNT 0) (REMAIN 1) (N 1) (M 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    0.5 TRUE IN PS
     STM (ANS 1) (COUNT 0) (REMAIN 1) (N 1) (M 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     5 TRUE IN PS
     STM (P M 1) (S COUNT 0) (ANS 1) (REMAIN 1) (N 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    0.5 TRUE IN PS
     STM (COUNT 1) (P M 1) (ANS 1) (REMAIN 1) (N 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    5.5 TRUE IN PS
     STM (M 0) (COUNT 1) (ANS 1) (REMAIN 1) (N 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     3 TRUE IN PS
     STM (S ANS 1) (COUNT 0) (REMAIN 0) (N 1) (M 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

    0.8 TRUE IN PS
     STM (ANS 2) (COUNT 0) (REMAIN 0) (N 1) (M 0)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

     4 TRUE IN PS

 2 WITH REMAINDER 0

     STM (M 0) (ANS 2) (REMAIN 0) (COUNT 0) (N 1)
        (ORDER 0 1 2 3 4 5 6 7 8 9)

 •display
  0.5. (S X3 0)  => (REP (S X3 0) (X3 1))
  0.8. (S X3 1)  => (REP (S X3 1) (X3 2))
  0.9. (S X3 2)  => (REP (S X3 2) (X3 3))
   1. (READY)  => (REP (READY) (COUNT 0))
      (DEP (ANS 0)) (ATTEND)
   2. (M X1) - (REMAIN)  => (DEP (REMAIN X1))
   3. (ANS) (COUNT X1) (REMAIN X3) (N X1) (M X2)
       => (REP (ANS) (S ANS)) (REP (X1) (0) 2)
      (REP (X3) (X2) 3)
   4. (M 0) (ANS X1) (REMAIN X2) - (S)  =>
      (SAY X1 WITH REMAINDER X2) (STOP)
   5. (M) (COUNT) - (S)  => (REP (M) (P M))
      (REP (COUNT) (S COUNT) 2)
  5.5. (P X3 1)  => (REP (P X3 1) (X3 0))
  5.8. (P X3 2)  => (REP (P X3 2) (X3 1))
  5.9. (P X3 3)  => (REP (P X3 3) (X3 2))
   6. (ORDER X1 X2) - (GO)  => (REP (X1 X2) (X2))
      (COND (P X3 X2))
      (ACTION (REP (P X3 X2) (X3 X1))) (PROD (GO))
      (COND (S X3 X1))
      (ACTION (REP (S X3 X1) (X3 X2))) (PROD (READY))
```

```
•memory display ps display
MEMORY MODE
     1. STM = (READY)

PS MODE
     1. (READY) (STIM X1) => (REM (READY))
        (PERCEIVE X1 ?)
     2. (READY) => (ATTEND STIM)
     3. (REPLY) - (RESP) => (ATTEND RESP)
     4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
     5. (USED X1) (WRONG X2) => (REP USED COND)
     6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)
     7. (X1 X2 ?) (RESP X3) (WRONG X4) =>
        (COND (X1 X2 ?))
        (ACTION (USED) (DEP (REPLY X3)) (SAY X3))
        (PROD (SAY X4)) (STOP)

•fire
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pax))
     STM: (STIM PAX) (READY)

     1 TRUE IN PS
     STM: (1 P ?) (3 X ?) (2 A ?) (STIM PAX)

     6 TRUE IN PS

?

OUTPUT FOR (ATTEND RESP) = (dep (resp con))
     STM: (RESP CON) (REPLY ?) (1 P ?) (3 X ?) (2 A ?)
        (STIM PAX)

     4 TRUE IN PS
     STM: (WRONG ?) (RESP CON) (1 P ?) (3 X ?) (2 A ?)
        (STIM PAX)

     7 TRUE IN PS
NOW INSERTING
(1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
ON LINE  5.5
     STM: (1 P ?) (RESP CON) (WRONG ?) (3 X ?) (2 A ?)
        (STIM PAX)

•display 5-6
     5. (USED X1) (WRONG X2) => (REP USED COND)
   5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
     6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim bek))
     STM: (STIM BEK) (READY)

     1 TRUE IN PS
     STM: (1 B ?) (3 K ?) (2 E ?) (STIM BEK)

     6 TRUE IN PS

?

OUTPUT FOR (ATTEND RESP) = (dep (resp luq))
     STM: (RESP LUQ) (REPLY ?) (1 B ?) (3 K ?) (2 E ?)
        (STIM BEK)

     4 TRUE IN PS
     STM: (WRONG ?) (RESP LUQ) (1 B ?) (3 K ?) (2 E ?)
        (STIM BEK)

     7 TRUE IN PS
NOW INSERTING
(1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
ON LINE  5.8
     STM: (1 B ?) (RESP LUQ) (WRONG ?) (3 K ?) (2 E ?)
        (STIM BEK)

•display 5-6
     5. (USED X1) (WRONG X2) => (REP USED COND)
   5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
   5.8. (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim cit))
     STM: (STIM CIT) (READY)

     1 TRUE IN PS
     STM: (1 C ?) (3 T ?) (2 I ?) (STIM CIT)

     6 TRUE IN PS

?

OUTPUT FOR (ATTEND RESP) = (dep (resp der))
     STM: (RESP DER) (REPLY ?) (1 C ?) (3 T ?) (2 I ?)
        (STIM CIT)

     4 TRUE IN PS
     STM: (WRONG ?) (RESP DER) (1 C ?) (3 T ?) (2 I ?)
        (STIM CIT)

     7 TRUE IN PS
NOW INSERTING
(1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
ON LINE  5.9
     STM: (1 C ?) (RESP DER) (WRONG ?) (3 T ?) (2 I ?)
        (STIM CIT)

•display 5-6
     5. (USED X1) (WRONG X2) => (REP USED COND)
   5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
   5.8. (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
   5.9. (1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
     6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim buk))
     STM: (STIM BUK) (READY)

     1 TRUE IN PS
     STM: (1 B ?) (3 K ?) (2 U ?) (STIM BUK)
```

```
    5.8 TRUE IN PS

  LUQ

      STM: (REPLY LUQ) (USED (1 B ?)) (3 K ?) (2 U ?)
          (STIM BUK)

      3 TRUE IN PS
  OUTPUT FOR (ATTEND RESP) • (dep (resp mab))
      STM: (RESP MAB) (REPLY LUQ) (USED (1 B ?)) (3 K ?)
          (2 U ?) (STIM BUK)

      4 TRUE IN PS
      STM: (WRONG LUQ) (RESP MAB) (USED (1 B ?)) (3 K ?)
          (2 U ?) (STIM BUK)

      5 TRUE IN PS
      STM: (COND (1 B ?)) (WRONG LUQ) (RESP MAB) (3 K ?)
          (2 U ?) (STIM BUK)

      7 TRUE IN PS
  NOW INSERTING
  (3 K ?) (1 B ?) •> (USED) (DEP (REPLY MAB)) (SAY MAB)
  ON LINE  5.7
      STM: (3 K ?) (RESP MAB) (WRONG LUQ) (2 U ?)
          (STIM BUK)

  •display 5-6
      5. (USED X1) (WRONG X2) •> (REP USED COND)
     5.5. (1 P ?) •> (USED) (DEP (REPLY CON)) (SAY CON)
     5.7. (3 K ?) (1 B ?) •> (USED) (DEP (REPLY MAB))
         (SAY MAB)
     5.8. (1 B ?) •> (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     5.9. (1 C ?) •> (USED) (DEP (REPLY DER)) (SAY DER)
      6. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
         (ATTEND RESP)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim nal))
      STM: (STIM NAL) (READY)

      1 TRUE IN PS
      STM: (1 N ?) (3 L ?) (2 A ?) (STIM NAL)

      6 TRUE IN PS

  ?

  OUTPUT FOR (ATTEND RESP) • (dep (resp leq))
      STM: (RESP LEQ) (REPLY ?) (1 N ?) (3 L ?) (2 A ?)
          (STIM NAL)

      4 TRUE IN PS
      STM: (WRONG ?) (RESP LEQ) (1 N ?) (3 L ?) (2 A ?)
          (STIM NAL)

      7 TRUE IN PS
  NOW INSERTING
  (1 N ?) •> (USED) (DEP (REPLY LEQ)) (SAY LEQ)
  ON LINE  5.95
      STM: (1 N ?) (RESP LEQ) (WRONG ?) (3 L ?) (2 A ?)
          (STIM NAL)
```

```
  •display 5-6
      5. (USED X1) (WRONG X2) •> (REP USED COND)
     5.5. (1 P ?) •> (USED) (DEP (REPLY CON)) (SAY CON)
     5.7. (3 K ?) (1 B ?) •> (USED) (DEP (REPLY MAB))
         (SAY MAB)
     5.8. (1 B ?) •> (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     5.9. (1 C ?) •> (USED) (DEP (REPLY DER)) (SAY DER)
    5.95. (1 N ?) •> (USED) (DEP (REPLY LEQ)) (SAY LEQ)
      6. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
         (ATTEND RESP)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim reb))
      STM: (STIM REB) (READY)

      1 TRUE IN PS
      STM: (1 R ?) (3 B ?) (2 E ?) (STIM REB)

      6 TRUE IN PS

  ?

  OUTPUT FOR (ATTEND RESP) • (dep (resp mol))
      STM: (RESP MOL) (REPLY ?) (1 R ?) (3 B ?) (2 E ?)
          (STIM REB)

      4 TRUE IN PS
      STM: (WRONG ?) (RESP MOL) (1 R ?) (3 B ?) (2 E ?)
          (STIM REB)

      7 TRUE IN PS
  NOW INSERTING
  (1 R ?) •> (USED) (DEP (REPLY MOL)) (SAY MOL)
  ON LINE  5.98
      STM: (1 R ?) (RESP MOL) (WRONG ?) (3 B ?) (2 E ?)
          (STIM REB)

  •display 5-6
      5. (USED X1) (WRONG X2) •> (REP USED COND)
     5.5. (1 P ?) •> (USED) (DEP (REPLY CON)) (SAY CON)
     5.7. (3 K ?) (1 B ?) •> (USED) (DEP (REPLY MAB))
         (SAY MAB)
     5.8. (1 B ?) •> (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     5.9. (1 C ?) •> (USED) (DEP (REPLY DER)) (SAY DER)
    5.95. (1 N ?) •> (USED) (DEP (REPLY LEQ)) (SAY LEQ)
    5.98. (1 R ?) •> (USED) (DEP (REPLY MOL)) (SAY MOL)
      6. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
         (ATTEND RESP)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim noj))
      STM: (STIM NOJ) (READY)

      1 TRUE IN PS
      STM: (1 N ?) (3 J ?) (2 O ?) (STIM NOJ)

    5.95 TRUE IN PS

  LEQ
```

```
      STM: (REPLY LEQ) (USED (1 N ?)) (3 J ?) (2 O ?)
          (STIM NOJ)

      3 TRUE IN PS
  OUTPUT FOR (ATTEND RESP) • (dep (resp ped))
      STM: (RESP PED) (REPLY LEQ) (USED (1 N ?)) (3 J ?)
          (2 O ?) (STIM NOJ)

      4 TRUE IN PS
      STM: (WRONG LEQ) (RESP PED) (USED (1 N ?)) (3 J ?)
          (2 O ?) (STIM NOJ)

      5 TRUE IN PS
      STM: (COND (1 N ?)) (WRONG LEQ) (RESP PED) (3 J ?)
          (2 O ?) (STIM NOJ)

      7 TRUE IN PS
  NOW INSERTING
  (3 J ?) (1 N ?) => (USED) (DEP (REPLY PED)) (SAY PED)
  ON LINE 5.93
      STM: (3 J ?) (RESP PED) (WRONG LEQ) (2 O ?)
          (STIM NOJ)

  •display 5-6
       5. (USED X1) (WRONG X2) => (REP USED COND)
     5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
     5.7. (3 K ?) (1 B ?) => (USED) (DEP (REPLY MAB))
          (SAY MAB)
     5.8. (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     5.9. (1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
    5.93. (3 J ?) (1 N ?) => (USED) (DEP (REPLY PED))
          (SAY PED)
    5.95. (1 N ?) => (USED) (DEP (REPLY LEQ)) (SAY LEQ)
    5.98. (1 R ?) => (USED) (DEP (REPLY MOL)) (SAY MOL)
       6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
          (ATTEND RESP)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim pax))
      STM: (STIM PAX) (READY)

      1 TRUE IN PS
      STM: (1 P ?) (3 X ?) (2 A ?) (STIM PAX)

    5.5 TRUE IN PS

  CON

      STM: (REPLY CON) (USED (1 P ?)) (3 X ?) (2 A ?)
          (STIM PAX)

      3 TRUE IN PS
  OUTPUT FOR (ATTEND RESP) • (dep (resp con))
      STM: (RESP CON) (REPLY CON) (USED (1 P ?)) (3 X ?)
          (2 A ?) (STIM PAX)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim bek))
      STM: (STIM BEK) (READY)

      1 TRUE IN PS
```

```
      STM: (1 B ?) (3 K ?) (2 E ?) (STIM BEK)

    5.7 TRUE IN PS

  MAB

      STM: (REPLY MAB) (USED (3 K ?)) (USED (1 B ?))
          (2 E ?) (STIM BEK)

      3 TRUE IN PS
  OUTPUT FOR (ATTEND RESP) • (dep (resp luq))
      STM: (RESP LUQ) (REPLY MAB) (USED (3 K ?))
          (USED (1 B ?)) (2 E ?) (STIM BEK)

      4 TRUE IN PS
      STM: (WRONG MAB) (RESP LUQ) (USED (3 K ?))
          (USED (1 B ?)) (2 E ?) (STIM BEK)

      5 TRUE IN PS
      STM: (COND (3 K ?)) (WRONG MAB) (RESP LUQ)
          (USED (1 B ?)) (2 E ?) (STIM BEK)

      5 TRUE IN PS
      STM: (COND (1 B ?)) (WRONG MAB) (COND (3 K ?))
          (RESP LUQ) (2 E ?) (STIM BEK)

      7 TRUE IN PS
  NOW INSERTING
  (2 E ?) (1 B ?) (3 K ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
  ON LINE 5.6
      STM: (2 E ?) (RESP LUQ) (WRONG MAB) (STIM BEK)

  •display 5-6
       5. (USED X1) (WRONG X2) => (REP USED COND)
     5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
     5.6. (2 E ?) (1 B ?) (3 K ?) => (USED)
          (DEP (REPLY LUQ)) (SAY LUQ)
     5.7. (3 K ?) (1 B ?) => (USED) (DEP (REPLY MAB))
          (SAY MAB)
     5.8. (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
     5.9. (1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
    5.93. (3 J ?) (1 N ?) => (USED) (DEP (REPLY PED))
          (SAY PED)
    5.95. (1 N ?) => (USED) (DEP (REPLY LEQ)) (SAY LEQ)
    5.98. (1 R ?) => (USED) (DEP (REPLY MOL)) (SAY MOL)
       6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
          (ATTEND RESP)

  •initialize fire
  INITIALIZED
      2 TRUE IN PS
  OUTPUT FOR (ATTEND STIM) • (dep (stim cit))
      STM: (STIM CIT) (READY)

      1 TRUE IN PS
      STM: (1 C ?) (3 T ?) (2 I ?) (STIM CIT)

    5.9 TRUE IN PS

  DER

      STM: (REPLY DER) (USED (1 C ?)) (3 T ?) (2 I ?)
          (STIM CIT)

      3 TRUE IN PS
```

```
OUTPUT FOR (ATTEND RESP) = (dep (resp der))
     STM (RESP DER) (REPLY DER) (USED (1 C ?)) (3 T ?)
        (2 I ?) (STIM CIT)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim buk))
     STM: (STIM BUK) (READY)

     1 TRUE IN PS
     STM: (1 B ?) (3 K ?) (2 U ?) (STIM BUK)

   5.7 TRUE IN PS

MAB

     STM: (REPLY MAB) (USED (3 K ?)) (USED (1 B ?))
        (2 U ?) (STIM BUK)

     3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp mab))
     STM (RESP MAB) (REPLY MAB) (USED (3 K ?))
        (USED (1 B ?)) (2 U ?) (STIM BUK)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim nal))
     STM: (STIM NAL) (READY)

     1 TRUE IN PS
     STM (1 N ?) (3 L ?) (2 A ?) (STIM NAL)

   5.95 TRUE IN PS

LEQ

     STM (REPLY LEQ) (USED (1 N ?)) (3 L ?) (2 A ?)
        (STIM NAL)

     3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp leq))
     STM: (RESP LEQ) (REPLY LEQ) (USED (1 N ?)) (3 L ?)
        (2 A ?) (STIM NAL)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim reb))
     STM: (STIM REB) (READY)

     1 TRUE IN PS
     STM: (1 R ?) (3 B ?) (2 E ?) (STIM REB)

   5.98 TRUE IN PS

MOL

     STM: (REPLY MOL) (USED (1 R ?)) (3 B ?) (2 E ?)
        (STIM REB)

     3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp mol))
     STM: (RESP MOL) (REPLY MOL) (USED (1 R ?)) (3 B ?)
        (2 E ?) (STIM REB)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim noj))
     STM: (STIM NOJ) (READY)

     1 TRUE IN PS
     STM: (1 N ?) (3 J ?) (2 O ?) (STIM NOJ)

   5.93 TRUE IN PS

PED

     STM: (REPLY PED) (USED (3 J ?)) (USED (1 N ?))
        (2 O ?) (STIM NOJ)

     3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp ped))
     STM (RESP PED) (REPLY PED) (USED (3 J ?))
        (USED (1 N ?)) (2 O ?) (STIM NOJ)

•display 5-6
     5. (USED X1) (WRONG X2) => (REP USED COND)
    5.5 (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
    5.6 (2 E ?) (1 B ?) (3 K ?) => (USED)
        (DEP (REPLY LUQ)) (SAY LUQ)
    5.7 (3 K ?) (1 B ?) => (USED) (DEP (REPLY MAB))
        (SAY MAB)
    5.8 (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
    5.9 (1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
   5.93 (3 J ?) (1 N ?) => (USED) (DEP (REPLY PED))
        (SAY PED)
   5.95 (1 N ?) => (USED) (DEP (REPLY LEQ)) (SAY LEQ)
   5.98 (1 R ?) => (USED) (DEP (REPLY MOL)) (SAY MOL)
      6 - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pax))
     STM: (STIM PAX) (READY)

     1 TRUE IN PS
     STM: (1 P ?) (3 X ?) (2 A ?) (STIM PAX)

   5.5 TRUE IN PS

CON

     STM: (REPLY CON) (USED (1 P ?)) (3 X ?) (2 A ?)
        (STIM PAX)

     3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp con))
     STM (RESP CON) (REPLY CON) (USED (1 P ?)) (3 X ?)
        (2 A ?) (STIM PAX)

•initialize fire
INITIALIZED
     2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim bek))
     STM: (STIM BEK) (READY)
    1 TRUE IN PS
    STM: (1 B ?) (3 K ?) (2 E ?) (STIM BEK)

  5.6 TRUE IN PS

LUQ

    STM: (REPLY LUQ) (USED (2 E ?)) (USED (1 B ?))
        (USED (3 K ?)) (STIM BEK)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp luq))
    STM: (RESP LUQ) (REPLY LUQ) (USED (2 E ?))
        (USED (1 B ?)) (USED (3 K ?)) (STIM BEK)

*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim cit))
    STM: (STIM CIT) (READY)

    1 TRUE IN PS
    STM: (1 C ?) (3 T ?) (2 I ?) (STIM CIT)

  5.9 TRUE IN PS

DER

    STM: (REPLY DER) (USED (1 C ?)) (3 T ?) (2 I ?)
        (STIM CIT)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp der))
    STM: (RESP DER) (REPLY DER) (USED (1 C ?)) (3 T ?)
        (2 I ?) (STIM CIT)

*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim buk))
    STM: (STIM BUK) (READY)

    1 TRUE IN PS
    STM: (1 B ?) (3 K ?) (2 U ?) (STIM BUK)

  5.7 TRUE IN PS

MAB

    STM: (REPLY MAB) (USED (3 K ?)) (USED (1 B ?))
        (2 U ?) (STIM BUK)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp mab))
    STM: (RESP MAB) (REPLY MAB) (USED (3 K ?))
        (USED (1 B ?)) (2 U ?) (STIM BUK)

*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim nal))
    STM: (STIM NAL) (READY)

    1 TRUE IN PS
    STM: (1 N ?) (3 L ?) (2 A ?) (STIM NAL)

  5.95 TRUE IN PS

LEQ

    STM: (REPLY LEQ) (USED (1 N ?)) (3 L ?) (2 A ?)
        (STIM NAL)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp leq))
    STM: (RESP LEQ) (REPLY LEQ) (USED (1 N ?)) (3 L ?)
        (2 A ?) (STIM NAL)

*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim reb))
    STM: (STIM REB) (READY)

    1 TRUE IN PS
    STM: (1 R ?) (3 B ?) (2 E ?) (STIM REB)

  5.98 TRUE IN PS

MOL

    STM: (REPLY MOL) (USED (1 R ?)) (3 B ?) (2 E ?)
        (STIM REB)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp mol))
    STM: (RESP MOL) (REPLY MOL) (USED (1 R ?)) (3 B ?)
        (2 E ?) (STIM REB)

*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim noj))
    STM: (STIM NOJ) (READY)

    1 TRUE IN PS
    STM: (1 N ?) (3 J ?) (2 O ?) (STIM NOJ)

  5.93 TRUE IN PS

PED

    STM: (REPLY PED) (USED (3 J ?)) (USED (1 N ?))
        (2 O ?) (STIM NOJ)

    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp ped))
    STM: (RESP PED) (REPLY PED) (USED (3 J ?))
        (USED (1 N ?)) (2 O ?) (STIM NOJ)

*display
    1. (READY) (STIM X1) => (REM (READY))
       (PERCEIVE X1 ?)
    2. (READY) => (ATTEND STIM)
    3. (REPLY) - (RESP) => (ATTEND RESP)
    4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
    5. (USED X1) (WRONG X2) => (REP USED COND)
  5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
  5.6. (2 E ?) (1 B ?) (3 K ?) => (USED)
```

```
      (DEP (REPLY LUQ)) (SAY LUQ)
 5.7. (3 K ?) (1 B ?) => (USED) (DEP (REPLY MAB))
      (SAY MAB)
 5.8. (1 B ?) => (USED) (DEP (REPLY LUQ)) (SAY LUQ)
 5.9. (1 C ?) => (USED) (DEP (REPLY DER)) (SAY DER)
5.93. (3 J ?) (1 N ?) => (USED) (DEP (REPLY PED))
      (SAY PED)
5.95. (1 N ?) => (USED) (DEP (REPLY LEQ)) (SAY LEQ)
5.98. (1 R ?) => (USED) (DEP (REPLY MOL)) (SAY MOL)
   6. - (RESP) => (DEP (REPLY ?)) (SAY ?)
      (ATTEND RESP)
   7. (X1 X2 ?) (RESP X3) (WRONG X4) =>
      (COND (X1 X2 ?))
      (ACTION (USED) (DEP (REPLY X3)) (SAY X3))
      (PROD (SAY X4)) (STOP)
```

APPENDIX IV. Trace of EPAM2 Production System.

```
•memory display ps display
MEMORY MODE
    1 STM • (READY)

PS MODE
     1. (READY) (STIM X1) •> (REM (READY))
        (PERCEIVE X1 ?)
     2. (READY) •> (ATTEND STIM)
     3. (REPLY) - (RESP) •> (ATTEND RESP)
     4. (REPLY X1) - (RESP X1) •> (REP REPLY WRONG)
     5. (REPLY X1) (RESP X1) •> (STOP)
     6. (USED) (TEST X1) - (TEST X2) •>
        (REP USED USED•)
     7. (TEST X1) (TEST X2) (X3 X4 ?) •>
        (REM (X3 X4 ?))
     8. (TEST X1) (TEST X2) - (R-GEN) •>
        (DEP (REPLY X1) (R-GEN)) (SAY X1)
     9. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)
    10. (RESP X1) - (X2 X3 RESP) •> (PERCEIVE X1 RESP)
    11. (WRONG) (TEST X1) (STIM X1) - (R-GEN) •>
        (DEP (R-GEN))
    12. (OLD X1) (R-GEN) •> (REP OLD COND)
        (DEP (HOLD X1))
    13. (USED X1) (USED•) (R-GEN) •> (REP USED COND)
        (DEP (HOLD X1))
    14. (R-GEN) (COND (X1 X2 ?)) (X1 X2 RESP) •>
        (REM (X1 X2 RESP))
    15. (X1 X2 RESP) (RESP X3) (WRONG X4) - (DONE)
        •> (COND (X1 X2 ?))
        (ACTION (OLD) (DEP (REPLY X3)) (SAY X3))
        (PROD (SAY X4) (TEST X4)) (DEP (DONE))
    16. (USED• X1) •> (REP USED• COND)
    17. (OLD) (DONE) - (TEST) •> (REP OLD COND)
    18. (R-GEN) (HOLD (X1 X2 ?)) •>
        (REM (HOLD (X1 X2 ?))) (ACTION (DEP (X1 X2 ?)))
    19. (R-GEN) (X1 X2 RESP) (STIM X3) (WRONG X4) •>
        (ACTION (DEP (TEST X3)))
        (ACTION (USED) (DEP (X1 X2 ?)))
        (PROD (DEP (TEST X3))) (STOP)
    20. (X1 X2 ?) (X3 X4 RESP) (STIM X5) (WRONG X6)
        •> (COND (X1 X2 ?))
        (ACTION (USED) (DEP (X3 X4 ?)) (DEP (TEST X5)))
        (PROD (SAY X6)) (STOP)

•fire
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim pax))
 1 9

?

OUTPUT FOR (ATTEND RESP) • (dep (resp con))
 4 10 15
NOW INSERTING
(1 C ?) •> (OLD) (DEP (REPLY CON)) (SAY CON)
ON LINE  8.5
 20
NOW INSERTING
(1 P ?) •> (USED) (DEP (1 C ?)) (DEP (TEST PAX))
ON LINE  8.8

•display 8-9
    8. (TEST X1) (TEST X2) - (R-GEN) •>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
  8.5 (1 C ?) •> (OLD) (DEP (REPLY CON)) (SAY CON)
  8.8 (1 P ?) •> (USED) (DEP (1 C ?))
      (DEP (TEST PAX))
    9. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim bek))
 1 9

?

OUTPUT FOR (ATTEND RESP) • (dep (resp luq))
 4 10 15
NOW INSERTING
(1 L ?) •> (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
ON LINE  8.9
 20
NOW INSERTING
(1 B ?) •> (USED) (DEP (1 L ?)) (DEP (TEST BEK))
ON LINE  8.95

•display 8-9
    8. (TEST X1) (TEST X2) - (R-GEN) •>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
  8.5 (1 C ?) •> (OLD) (DEP (REPLY CON)) (SAY CON)
  8.8 (1 P ?) •> (USED) (DEP (1 C ?))
      (DEP (TEST PAX))
  8.9 (1 L ?) •> (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
 8.95 (1 B ?) •> (USED) (DEP (1 L ?))
      (DEP (TEST BEK))
    9. - (RESP) •> (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim cit))
 1 8.5

CON

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp der))
 4 10 15
NOW INSERTING
(1 D ?) •> (OLD) (DEP (REPLY DER)) (SAY DER)
ON LINE  8.3
 17 20
NOW INSERTING
(3 T ?) (1 C ?) •> (USED) (DEP (1 D ?)) (DEP (TEST CIT))
ON LINE  8.4

•display 8-9
    8. (TEST X1) (TEST X2) - (R-GEN) •>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
  8.3 (1 D ?) •> (OLD) (DEP (REPLY DER)) (SAY DER)
  8.4 (3 T ?) (1 C ?) •> (USED) (DEP (1 D ?))
      (DEP (TEST CIT))
  8.5 (1 C ?) •> (OLD) (DEP (REPLY CON)) (SAY CON)
  8.8 (1 P ?) •> (USED) (DEP (1 C ?))
      (DEP (TEST PAX))
  8.9 (1 L ?) •> (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
```

```
   8.95. (1 B ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST BEK))
     9. - (RESP)  => (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim buk))
 1 8.95 6 8.9

LUQ

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp mab))
 4 10 15
NOW INSERTING
(1 M ?)  => (OLD) (DEP (REPLY MAB)) (SAY MAB)
ON LINE  8.85
 16 20
NOW INSERTING
(3 K ?) (1 B ?)  => (USED) (DEP (1 M ?)) (DEP (TEST BUK))
ON LINE  8.88

•display 8-9
     8. (TEST X1) (TEST X2) - (R-GEN)  =>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
   8.3. (1 D ?)  => (OLD) (DEP (REPLY DER)) (SAY DER)
   8.4. (3 T ?) (1 C ?)  => (USED) (DEP (1 D ?))
       (DEP (TEST CIT))
   8.5. (1 C ?)  => (OLD) (DEP (REPLY CON)) (SAY CON)
   8.8. (1 P ?)  => (USED) (DEP (1 C ?))
       (DEP (TEST PAX))
  8.85. (1 M ?)  => (OLD) (DEP (REPLY MAB)) (SAY MAB)
  8.88. (3 K ?) (1 B ?)  => (USED) (DEP (1 M ?))
       (DEP (TEST BUK))
   8.9. (1 L ?)  => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.95. (1 B ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST BEK))
     9. - (RESP)  => (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim nal))
 1 9

?

OUTPUT FOR (ATTEND RESP) = (dep (resp leq))
 4 10 15
NOW INSERTING
(1 L ?)  => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
ON LINE  8.98
 20
NOW INSERTING
(1 N ?)  => (USED) (DEP (1 L ?)) (DEP (TEST NAL))
ON LINE  8.99

•display 8-9
     8. (TEST X1) (TEST X2) - (R-GEN)  =>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
   8.3. (1 D ?)  => (OLD) (DEP (REPLY DER)) (SAY DER)
   8.4. (3 T ?) (1 C ?)  => (USED) (DEP (1 D ?))
       (DEP (TEST CIT))
   8.5. (1 C ?)  => (OLD) (DEP (REPLY CON)) (SAY CON)
   8.8. (1 P ?)  => (USED) (DEP (1 C ?))
       (DEP (TEST PAX))
  8.85. (1 M ?)  => (OLD) (DEP (REPLY MAB)) (SAY MAB)
  8.88. (3 K ?) (1 B ?)  => (USED) (DEP (1 M ?))
       (DEP (TEST BUK))
   8.9. (1 L ?)  => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.95. (1 B ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST BEK))
  8.98. (1 L ?)  => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.99. (1 N ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST NAL))
     9. - (RESP)  => (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim reb))
 1 9

?

OUTPUT FOR (ATTEND RESP) = (dep (resp mol))
 4 10 15
NOW INSERTING
(1 M ?)  => (OLD) (DEP (REPLY MOL)) (SAY MOL)
ON LINE 8.995
 20
NOW INSERTING
(1 R ?)  => (USED) (DEP (1 M ?)) (DEP (TEST REB))
ON LINE 8.998

•display 8-9
     8. (TEST X1) (TEST X2) - (R-GEN)  =>
       (DEP (REPLY X1) (R-GEN)) (SAY X1)
   8.3. (1 D ?)  => (OLD) (DEP (REPLY DER)) (SAY DER)
   8.4. (3 T ?) (1 C ?)  => (USED) (DEP (1 D ?))
       (DEP (TEST CIT))
   8.5. (1 C ?)  => (OLD) (DEP (REPLY CON)) (SAY CON)
   8.8. (1 P ?)  => (USED) (DEP (1 C ?))
       (DEP (TEST PAX))
  8.85. (1 M ?)  => (OLD) (DEP (REPLY MAB)) (SAY MAB)
  8.88. (3 K ?) (1 B ?)  => (USED) (DEP (1 M ?))
       (DEP (TEST BUK))
   8.9. (1 L ?)  => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.95. (1 B ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST BEK))
  8.98. (1 L ?)  => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.99. (1 N ?)  => (USED) (DEP (1 L ?))
       (DEP (TEST NAL))
 8.995. (1 M ?)  => (OLD) (DEP (REPLY MOL)) (SAY MOL)
 8.998. (1 R ?)  => (USED) (DEP (1 M ?))
       (DEP (TEST REB))
     9. - (RESP)  => (DEP (REPLY ?)) (SAY ?)
       (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim noj))
 1 8.99 6 8.9

LUQ
```

```
 3
OUTPUT FOR (ATTEND RESP) = (dep (resp ped))
 4 10 15
NOW INSERTING
(1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
ON LINE 8.89
 16 20
NOW INSERTING
(3 J ?) (1 N ?) => (USED) (DEP (1 P ?)) (DEP (TEST NOJ))
ON LINE 8.895

•display 8-9
     8. (TEST X1) (TEST X2) - (R-GEN) =>
        (DEP (REPLY X1) (R-GEN)) (SAY X1)
   8.3. (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
   8.4. (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
        (DEP (TEST CIT))
   8.5. (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
   8.8. (1 P ?) => (USED) (DEP (1 C ?))
        (DEP (TEST PAX))
  8.85. (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
  8.88. (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
        (DEP (TEST BUK))
  8.89. (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
 8.895. (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
        (DEP (TEST NOJ))
   8.9. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.95. (1 B ?) => (USED) (DEP (1 L ?))
        (DEP (TEST BEK))
  8.98. (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.99. (1 N ?) => (USED) (DEP (1 L ?))
        (DEP (TEST NAL))
 8.995. (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
 8.998. (1 R ?) => (USED) (DEP (1 M ?))
        (DEP (TEST REB))
     9. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim pax))
 1 8.8 6 8.5

CON

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp con))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim bek))
 1 8.88 6 6 8.85

MAB

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp luq))
 4 10 15
NOW INSERTING
(1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
ON LINE 8.83
```

```
 16 16 20
NOW INSERTING
(2 E ?) (3 K ?) (1 B ?) => (USED) (DEP (1 L ?)) (DEP (TEST BEK))
ON LINE 8.84

•display 8-9
     8. (TEST X1) (TEST X2) - (R-GEN) =>
        (DEP (REPLY X1) (R-GEN)) (SAY X1)
   8.3. (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
   8.4. (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
        (DEP (TEST CIT))
   8.5. (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
   8.8. (1 P ?) => (USED) (DEP (1 C ?))
        (DEP (TEST PAX))
  8.83. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.84. (2 E ?) (3 K ?) (1 B ?) => (USED)
        (DEP (1 L ?)) (DEP (TEST BEK))
  8.85. (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
  8.88. (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
        (DEP (TEST BUK))
  8.89. (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
 8.895. (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
        (DEP (TEST NOJ))
   8.9. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
  8.95. (1 B ?) => (USED) (DEP (1 L ?))
        (DEP (TEST BEK))
  8.98. (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.99. (1 N ?) => (USED) (DEP (1 L ?))
        (DEP (TEST NAL))
 8.995. (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
 8.998. (1 R ?) => (USED) (DEP (1 M ?))
        (DEP (TEST REB))
     9. - (RESP) => (DEP (REPLY ?)) (SAY ?)
        (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim cit))
 1 8.4 6 6 8.3

DER

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp der))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim buk))
 1 8.88 6 6 8.85

MAB

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp mab))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim nal))
 1 8.99 6 8.83
```

```
LUQ

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp leq))
 4 10 11 12 14 15
NOW INSERTING
(3 Q ?) (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
ON LINE 8.82
 16 18 19
NOW INSERTING
(1 N ?) => (USED) (DEP (3 Q ?)) (DEP (TEST NAL)) (DEP (1 L ?))
ON LINE 8.985

•display 8-9
      8. (TEST X1) (TEST X2) - (R-GEN) =>
         (DEP (REPLY X1) (R-GEN)) (SAY X1)
    8.3. (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
    8.4. (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
         (DEP (TEST CIT))
    8.5. (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
    8.8. (1 P ?) => (USED) (DEP (1 C ?))
         (DEP (TEST PAX))
   8.82. (3 Q ?) (1 L ?) => (OLD) (DEP (REPLY LEQ))
         (SAY LEQ)
   8.83. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
   8.84. (2 E ?) (3 K ?) (1 B ?) => (USED)
         (DEP (1 L ?)) (DEP (TEST BEK))
   8.85. (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
   8.88. (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
         (DEP (TEST BUK))
   8.89. (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
  8.895. (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
         (DEP (TEST NOJ))
    8.9. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
   8.95. (1 B ?) => (USED) (DEP (1 L ?))
         (DEP (TEST BEK))
   8.98. (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.985. (1 N ?) => (USED) (DEP (3 Q ?))
         (DEP (TEST NAL)) (DEP (1 L ?))
   8.99. (1 N ?) => (USED) (DEP (1 L ?))
         (DEP (TEST NAL))
  8.995. (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
  8.998. (1 R ?) => (USED) (DEP (1 M ?))
         (DEP (TEST REB))
      9. - (RESP) => (DEP (REPLY ?)) (SAY ?)
         (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim reb))
 1 8.998 6 8.85

MAB

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp mol))
 4 10 11 12 14 15
NOW INSERTING
(3 L ?) (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
ON LINE 8.845
 16 18 19
NOW INSERTING
(1 R ?) => (USED) (DEP (3 L ?)) (DEP (TEST REB)) (DEP (1 M ?))
ON LINE 8.997
```

```
•display 8-9
      8. (TEST X1) (TEST X2) - (R-GEN) =>
         (DEP (REPLY X1) (R-GEN)) (SAY X1)
    8.3. (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
    8.4. (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
         (DEP (TEST CIT))
    8.5. (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
    8.8. (1 P ?) => (USED) (DEP (1 C ?))
         (DEP (TEST PAX))
   8.82. (3 Q ?) (1 L ?) => (OLD) (DEP (REPLY LEQ))
         (SAY LEQ)
   8.83. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
   8.84. (2 E ?) (3 K ?) (1 B ?) => (USED)
         (DEP (1 L ?)) (DEP (TEST BEK))
  8.845. (3 L ?) (1 M ?) => (OLD) (DEP (REPLY MOL))
         (SAY MOL)
   8.85. (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
   8.88. (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
         (DEP (TEST BUK))
   8.89. (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
  8.895. (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
         (DEP (TEST NOJ))
    8.9. (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
   8.95. (1 B ?) => (USED) (DEP (1 L ?))
         (DEP (TEST BEK))
   8.98. (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
  8.985. (1 N ?) => (USED) (DEP (3 Q ?))
         (DEP (TEST NAL)) (DEP (1 L ?))
   8.99. (1 N ?) => (USED) (DEP (1 L ?))
         (DEP (TEST NAL))
  8.995. (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
  8.997. (1 R ?) => (USED) (DEP (3 L ?))
         (DEP (TEST REB)) (DEP (1 M ?))
  8.998. (1 R ?) => (USED) (DEP (1 M ?))
         (DEP (TEST REB))
      9. - (RESP) => (DEP (REPLY ?)) (SAY ?)
         (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) = (dep (stim noj))
 1 8.895 6 6 8.8 7 7 8

PAX

 3
OUTPUT FOR (ATTEND RESP) = (dep (resp ped))
 4 10 13 14 15
NOW INSERTING
(3 D ?) (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
ON LINE  8.7
 16 16 18 19
NOW INSERTING
(3 J ?) (1 N ?) => (USED) (DEP (3 D ?)) (DEP (TEST NOJ)) (DEP (1 P ?))
ON LINE 8.893

•display 8-9
      8. (TEST X1) (TEST X2) - (R-GEN) =>
         (DEP (REPLY X1) (R-GEN)) (SAY X1)
    8.3. (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
    8.4. (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
         (DEP (TEST CIT))
    8.5. (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
  8.7 (3 D ?) (1 P ?) => (OLD) (DEP (REPLY PED))
      (SAY PED)
  8.8 (1 P ?) => (USED) (DEP (1 C ?))
      (DEP (TEST PAX))
 8.82 (3 Q ?) (1 L ?) => (OLD) (DEP (REPLY LEQ))
      (SAY LEQ)
 8.83 (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
 8.84 (2 E ?) (3 K ?) (1 B ?) => (USED)
      (DEP (1 L ?)) (DEP (TEST BEK))
8.845 (3 L ?) (1 M ?) => (OLD) (DEP (REPLY MOL))
      (SAY MOL)
 8.85 (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
 8.88 (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
      (DEP (TEST BUK))
 8.89 (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
8.893 (3 J ?) (1 N ?) => (USED) (DEP (3 D ?))
      (DEP (TEST NOJ)) (DEP (1 P ?))
8.895 (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
      (DEP (TEST NOJ))
  8.9 (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
 8.95 (1 B ?) => (USED) (DEP (1 L ?))
      (DEP (TEST BEK))
 8.98 (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
8.985 (1 N ?) => (USED) (DEP (3 Q ?))
      (DEP (TEST NAL)) (DEP (1 L ?))
 8.99 (1 N ?) => (USED) (DEP (1 L ?))
      (DEP (TEST NAL))
8.995 (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
8.997 (1 R ?) => (USED) (DEP (3 L ?))
      (DEP (TEST REB)) (DEP (1 M ?))
8.998 (1 R ?) => (USED) (DEP (1 M ?))
      (DEP (TEST REB))
   9. - (RESP) => (DEP (REPLY ?)) (SAY ?)
      (ATTEND RESP)

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim pax))
 1 8.8 6 8.5

CON

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp con))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim bek))
 1 8.84 6 6 6 8.83

LUQ

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp luq))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim cit))
 1 8.4 6 6 8.3

DER

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp der))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim buk))
 1 8.88 6 6 8.85

MAB

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp mab))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim nal))
 1 8.985 6 8.82

LEQ

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp leq))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim reb))
 1 8.997 6 8.845

MOL

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp mol))
 5

•initialize fire
INITIALIZED
 2
OUTPUT FOR (ATTEND STIM) • (dep (stim noj))
 1 8.893 6 6 8.7

PED

 3
OUTPUT FOR (ATTEND RESP) • (dep (resp ped))
 5

•display
    1. (READY) (STIM X1) => (REM (READY))
       (PERCEIVE X1 ?)
    2. (READY) => (ATTEND STIM)
    3. (REPLY) - (RESP) => (ATTEND RESP)
    4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
    5. (REPLY X1) (RESP X1) => (STOP)
    6. (USED) (TEST X1) - (TEST X2) =>
       (REP USED USED.)
    7. (TEST X1) (TEST X2) (X3 X4 ?) =>
       (REM (X3 X4 ?))
 8.   (TEST X1) (TEST X2) - (R-GEN) =>
      (DEP (REPLY X1) (R-GEN)) (SAY X1)
 8.3  (1 D ?) => (OLD) (DEP (REPLY DER)) (SAY DER)
 8.4  (3 T ?) (1 C ?) => (USED) (DEP (1 D ?))
      (DEP (TEST CIT))
 8.5  (1 C ?) => (OLD) (DEP (REPLY CON)) (SAY CON)
 8.7  (3 D ?) (1 P ?) => (OLD) (DEP (REPLY PED))
      (SAY PED)
 8.8  (1 P ?) => (USED) (DEP (1 C ?))
      (DEP (TEST PAX))
8.82  (3 Q ?) (1 L ?) => (OLD) (DEP (REPLY LEQ))
      (SAY LEQ)
8.83  (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
8.84  (2 E ?) (3 K ?) (1 B ?) => (USED)
      (DEP (1 L ?)) (DEP (TEST BEK))
8.845 (3 L ?) (1 M ?) => (OLD) (DEP (REPLY MOL))
      (SAY MOL)
8.85  (1 M ?) => (OLD) (DEP (REPLY MAB)) (SAY MAB)
8.88  (3 K ?) (1 B ?) => (USED) (DEP (1 M ?))
      (DEP (TEST BUK))
8.89  (1 P ?) => (OLD) (DEP (REPLY PED)) (SAY PED)
8.893 (3 J ?) (1 N ?) => (USED) (DEP (3 D ?))
      (DEP (TEST NOJ)) (DEP (1 P ?))
8.895 (3 J ?) (1 N ?) => (USED) (DEP (1 P ?))
      (DEP (TEST NOJ))
 8.9  (1 L ?) => (OLD) (DEP (REPLY LUQ)) (SAY LUQ)
8.95  (1 B ?) => (USED) (DEP (1 L ?))
      (DEP (TEST BEK))
8.98  (1 L ?) => (OLD) (DEP (REPLY LEQ)) (SAY LEQ)
8.985 (1 N ?) => (USED) (DEP (3 Q ?))
      (DEP (TEST NAL)) (DEP (1 L ?))
8.99  (1 N ?) => (USED) (DEP (1 L ?))
      (DEP (TEST NAL))
8.995 (1 M ?) => (OLD) (DEP (REPLY MOL)) (SAY MOL)
8.997 (1 R ?) => (USED) (DEP (3 L ?))
      (DEP (TEST REB)) (DEP (1 M ?))
8.998 (1 R ?) => (USED) (DEP (1 M ?))
      (DEP (TEST REB))
 9.   - (RESP) => (DEP (REPLY ?)) (SAY ?)
      (ATTEND RESP)
10.   (RESP X1) - (X2 X3 RESP) => (PERCEIVE X1 RESP)
11.   (WRONG) (TEST X1) (STIM X1) - (R-GEN) =>
      (DEP (R-GEN))
12.   (OLD X1) (R-GEN) => (REP OLD COND)
      (DEP (HOLD X1))
13.   (USED X1) (USED·) (R-GEN) => (REP USED COND)
      (DEP (HOLD X1))
14.   (R-GEN) (COND (X1 X2 ?)) (X1 X2 RESP) =>
      (REM (X1 X2 RESP))
15.   (X1 X2 RESP) (RESP X3) (WRONG X4) - (DONE)
      => (COND (X1 X2 ?))
      (ACTION (OLD) (DEP (REPLY X3)) (SAY X3))
      (PROD (SAY X4) (TEST X4)) (DEP (DONE))
16.   (USED· X1) => (REP USED· COND)
17.   (OLD) (DONE) - (TEST) => (REP OLD COND)
18.   (R-GEN) (HOLD (X1 X2 ?)) =>
      (REM (HOLD (X1 X2 ?))) (ACTION (DEP (X1 X2 ?)))
19.   (R-GEN) (X1 X2 RESP) (STIM X3) (WRONG X4) =>
      (ACTION (DEP (TEST X3)))
      (ACTION (USED) (DEP (X1 X2 ?)))
      (PROD (DEP (TEST X3))) (STOP)
20.   (X1 X2 ?) (X3 X4 RESP) (STIM X5) (WRONG X6)
      => (COND (X1 X2 ?))
      (ACTION (USED) (DEP (X3 X4 ?)) (DEP (TEST X5)))
      (PROD (SAY X6)) (STOP)
```

```
*memory display ps display
MEMORY MODE
    1. STM = (READY)

PS MODE
    1. (READY) (SERIES X1) => (REP READY CONT)
       (OBSERVE X1 ?)
    2. (READY) => (ATTEND SERIES)
    3. (X1 ?) - (LOC) => (COND (1 X4 ?))
       (ACTION (DEP (NEXT X1))) (PROD END)
       (DEP (LOC X1))
    4. (0 X1 ?) => (SUCC)
    5. (ERROR) (SERIES X1) (LOC X2) - (X3 ?) =>
       (CLEAR (SERIES X1) (LOC X2)) (DEP (READY))
    6. (NEXT X1) - (X2 ?) => (SAY X1)
       (DEP (MATCH) (X1 ?)) (STOP)
    7. (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
       - (MATCH) => (DEP (MATCH))
    8. (USED) - (MATCH) - (ERROR) => (DEP (ERROR))
    9. (USED (X1 X2 ?)) (NEXT) => (REP USED OLD)
   10. (X1 X2 ?) (NEXT) - (DONE) =>
       (REP (X1 X2 ?) (OLD (X1 X2 ?))) (DEP (DONE))
   11. (OLD (X1 X2 ?)) => (REP OLD COND)
       (DEP (X1 X2 ?))
   12. (NEXT X1) (MATCH) (SERIES X2) =>
       (REP (NEXT X1) CONT) (REM (MATCH) (DONE))
       (PROD (SERIES X2))
   13. (LOC X1) (NEXT X2)
       (ACTION (USED) (DEP (NEXT X3))) => (REP X1 X3)
       (REP (NEXT X2) CONT 2) (REM (DONE))
       (PROD (NEXT X1))
   14. (X1 ?) (CONT) (X2 ?) => (REP X1 (0 X1))
       (REM (CONT)) (ACTION (USED) (DEP (NEXT X2)))
   15. (X1 ?) (CONT) => (REP X1 (0 X1)) (REM (CONT))

*
*fire
    2 TRUE IN PS
OUTPUT FOR (ATTEND SERIES) = (dep (series abab))
    STM: (SERIES ABAB) (READY)

    1 TRUE IN PS
    STM: (A ?) (B ?) (A ?) (B ?) (CONT) (SERIES ABAB)

    3 TRUE IN PS
NOW INSERTING
(1 X4 ?) => (DEP (NEXT A))
ON LINE   16
    STM: (LOC A) (A ?) (B ?) (A ?) (B ?) (CONT)
       (SERIES ABAB)

    14 TRUE IN PS
    STM: (ACTION (USED) (DEP (NEXT B))) (0 A ?) (B ?)
       (LOC A) (A ?) (B ?) (SERIES ABAB)

    4 TRUE IN PS
    STM: (1 A ?) (ACTION (USED) (DEP (NEXT B))) (B ?)
       (LOC A) (A ?) (B ?) (SERIES ABAB)

    16 TRUE IN PS
    STM: (NEXT A) (1 A ?)
       (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
       (A ?) (B ?) (SERIES ABAB)

    10 TRUE IN PS
    STM: (DONE) (OLD (1 A ?)) (NEXT A)
       (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
       (A ?) (B ?) (SERIES ABAB)

    11 TRUE IN PS
    STM: (1 A ?) (COND (1 A ?)) (DONE) (NEXT A)
       (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
       (A ?) (B ?) (SERIES ABAB)

    13 TRUE IN PS
NOW INSERTING
(1 A ?) => (USED) (DEP (NEXT B))
ON LINE  15.5
    STM: (LOC B) (CONT) (1 A ?) (B ?) (A ?) (B ?)
       (SERIES ABAB)

    14 TRUE IN PS
    STM: (ACTION (USED) (DEP (NEXT A))) (0 B ?) (A ?)
       (LOC B) (1 A ?) (B ?) (SERIES ABAB)

    4 TRUE IN PS
    STM: (1 B ?) (ACTION (USED) (DEP (NEXT A))) (A ?)
       (LOC B) (2 A ?) (B ?) (SERIES ABAB)

    16 TRUE IN PS
    STM: (NEXT A) (1 B ?)
       (ACTION (USED) (DEP (NEXT A))) (A ?) (LOC B)
       (2 A ?) (B ?) (SERIES ABAB)

    10 TRUE IN PS
    STM: (DONE) (OLD (1 B ?)) (NEXT A)
       (ACTION (USED) (DEP (NEXT A))) (A ?) (LOC B)
       (2 A ?) (B ?) (SERIES ABAB)

    11 TRUE IN PS
    STM: (1 B ?) (COND (1 B ?)) (DONE) (NEXT A)
       (ACTION (USED) (DEP (NEXT A))) (A ?) (LOC B)
       (2 A ?) (B ?) (SERIES ABAB)

    13 TRUE IN PS
NOW INSERTING
(1 B ?) => (USED) (DEP (NEXT A))
ON LINE  15.3
    STM: (LOC A) (CONT) (1 B ?) (A ?) (2 A ?) (B ?)
       (SERIES ABAB)

    14 TRUE IN PS
    STM: (ACTION (USED) (DEP (NEXT B))) (0 A ?) (B ?)
       (LOC A) (1 B ?) (2 A ?) (SERIES ABAB)

    4 TRUE IN PS
    STM: (1 A ?) (ACTION (USED) (DEP (NEXT B))) (B ?)
       (LOC A) (2 B ?) (3 A ?) (SERIES ABAB)

    15.5 TRUE IN PS
    STM: (NEXT B) (USED (1 A ?))
       (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
       (2 B ?) (3 A ?) (SERIES ABAB)

    7 TRUE IN PS
    STM: (MATCH) (NEXT B) (USED (1 A ?))
       (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
       (2 B ?) (3 A ?) (SERIES ABAB)

    9 TRUE IN PS
   STM: (OLD (1 A ?)) (NEXT B) (MATCH)
      (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
      (2 B ?) (3 A ?) (SERIES ABAB)

   10 TRUE IN PS
   STM: (DONE) (OLD (2 B ?)) (NEXT B) (OLD (1 A ?))
      (MATCH) (ACTION (USED) (DEP (NEXT B))) (B ?)
      (LOC A) (3 A ?) (SERIES ABAB)

   11 TRUE IN PS
   STM: (2 B ?) (COND (2 B ?)) (DONE) (NEXT B)
      (OLD (1 A ?)) (MATCH)
      (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
      (3 A ?) (SERIES ABAB)

   11 TRUE IN PS
   STM: (1 A ?) (COND (1 A ?)) (2 B ?) (COND (2 B ?))
      (DONE) (NEXT B) (MATCH)
      (ACTION (USED) (DEP (NEXT B))) (B ?) (LOC A)
      (3 A ?) (SERIES ABAB)

   12 TRUE IN PS
   STM: (CONT) (SERIES ABAB) (1 A ?) (2 B ?) (B ?)
      (LOC A) (3 A ?)

   15 TRUE IN PS
   STM: (0 B ?) (SERIES ABAB) (1 A ?) (2 B ?) (LOC A)
      (3 A ?)

   4 TRUE IN PS
   STM: (1 B ?) (SERIES ABAB) (2 A ?) (3 B ?) (LOC A)
      (4 A ?)

  15.3 TRUE IN PS
   STM: (NEXT A) (USED (1 B ?)) (SERIES ABAB) (2 A ?)
      (3 B ?) (LOC A) (4 A ?)

   6 TRUE IN PS

A

   STM: (A ?) (MATCH) (NEXT A) (USED (1 B ?))
      (SERIES ABAB) (2 A ?) (3 B ?) (LOC A) (4 A ?)

•
•display 15-16
   15 (X1 ?) (CONT) -> (REP X1 (0 X1)) (REM (CONT))
  153 (1 B ?) -> (USED) (DEP (NEXT A))
  155 (1 A ?) -> (USED) (DEP (NEXT B))
   16. (1 X4 ?) -> (DEP (NEXT A))

•fire
   9 TRUE IN PS
   STM: (OLD (1 B ?)) (NEXT A) (A ?) (MATCH)
      (SERIES ABAB) (2 A ?) (3 B ?) (LOC A) (4 A ?)

   10 TRUE IN PS
   STM: (DONE) (OLD (2 A ?)) (NEXT A) (OLD (1 B ?))
      (A ?) (MATCH) (SERIES ABAB) (3 B ?) (LOC A)
      (4 A ?)

   11 TRUE IN PS
   STM: (2 A ?) (COND (2 A ?)) (DONE) (NEXT A)
      (OLD (1 B ?)) (A ?) (MATCH) (SERIES ABAB)
      (3 B ?) (LOC A) (4 A ?)
```

```
   11 TRUE IN PS
   STM: (1 B ?) (COND (1 B ?)) (2 A ?) (COND (2 A ?))
      (DONE) (NEXT A) (A ?) (MATCH) (SERIES ABAB)
      (3 B ?) (LOC A) (4 A ?)

   12 TRUE IN PS
   STM: (CONT) (SERIES ABAB) (1 B ?) (2 A ?) (A ?)
      (3 B ?) (LOC A) (4 A ?)

   15 TRUE IN PS
   STM: (0 A ?) (SERIES ABAB) (1 B ?) (2 A ?) (3 B ?)
      (LOC A) (4 A ?)

   4 TRUE IN PS
   STM: (1 A ?) (SERIES ABAB) (2 B ?) (3 A ?) (4 B ?)
      (LOC A) (5 A ?)

  155 TRUE IN PS
   STM: (NEXT B) (USED (1 A ?)) (SERIES ABAB) (2 B ?)
      (3 A ?) (4 B ?) (LOC A) (5 A ?)

   6 TRUE IN PS

B

   STM: (B ?) (MATCH) (NEXT B) (USED (1 A ?))
      (SERIES ABAB) (2 B ?) (3 A ?) (4 B ?) (LOC A)
      (5 A ?)
```

APPENDIX VI. Trace of Simple Series Completion Production System on ABAACAABA.

```
•memory display ps display
MEMORY MODE
    1. STM = (READY)

PS MODE
    1. (READY) (SERIES X1) => (REP READY CONT)
       (OBSERVE X1 ?)
    2. (READY) => (ATTEND SERIES)
    3. (X1 ?) - (LOC) => (COND (1 X4 ?))
       (ACTION (DEP (NEXT X1))) (PROD END)
       (DEP (LOC X1))
    4. (0 X1 ?) => (SUCC)
    5. (ERROR) (SERIES X1) (LOC X2) - (X3 ?) =>
       (CLEAR (SERIES X1) (LOC X2)) (DEP (READY))
    6. (NEXT X1) - (X2 ?) => (SAY X1)
       (DEP (MATCH) (X1 ?)) (STOP)
    7. (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
       - (MATCH) => (DEP (MATCH))
    8. (USED) - (MATCH) - (ERROR) => (DEP (ERROR))
    9. (USED (X1 X2 ?)) (NEXT) => (REP USED OLD)
   10. (X1 X2 ?) (NEXT) - (DONE) =>
       (REP (X1 X2 ?) (OLD (X1 X2 ?))) (DEP (DONE))
   11. (OLD (X1 X2 ?)) => (REP OLD COND)
       (DEP (X1 X2 ?))
   12. (NEXT X1) (MATCH) (SERIES X2) =>
       (REP (NEXT X1) CONT) (REM (MATCH) (DONE))
       (PROD (SERIES X2))
   13. (LOC X1) (NEXT X2)
       (ACTION (USED) (DEP (NEXT X3))) => (REP X1 X3)
       (REP (NEXT X2) CONT 2) (REM (DONE))
       (PROD (NEXT X1))
   14. (X1 ?) (CONT) (X2 ?) => (REP X1 (0 X1))
       (REM (CONT)) (ACTION (USED) (DEP (NEXT X2)))
   15. (X1 ?) (CONT) => (REP X1 (0 X1)) (REM (CONT))

•
•fire
 2
OUTPUT FOR (ATTEND SERIES) = (dep (series abaacaaba))
 1 3
NOW INSERTING
(1 X4 ?) => (DEP (NEXT A))
ON LINE   16
 14 4 16 10 11 13
NOW INSERTING
(1 A ?) => (USED) (DEP (NEXT B))
ON LINE  15.5
 14 4 16 10 11 13
NOW INSERTING
(1 B ?) => (USED) (DEP (NEXT A))
ON LINE  15.3
 14 4 15.5 8 9 10 11 11 13
NOW INSERTING
(1 A ?) (2 B ?) => (USED) (DEP (NEXT A))
ON LINE  15.2
 14 4 15.5 9 10 11 11 13
NOW INSERTING
(1 A ?) (2 A ?) => (USED) (DEP (NEXT C))
ON LINE  15.1
 14 4 16 10 11 13
NOW INSERTING
(1 C ?) => (USED) (DEP (NEXT A))
ON LINE 15.05
 14 4 15.5 9 10 11 11 13
NOW INSERTING
(1 A ?) (2 C ?) => (USED) (DEP (NEXT A))
ON LINE 15.03
 14 4 15.1 9 9 10 11 11 11 13
NOW INSERTING
(1 A ?) (2 A ?) (3 C ?) => (USED) (DEP (NEXT B))
ON LINE 15.02
 14 4 15.3 7 9 10 11 11 12 15 4 5 1 14 4 15.5 7 9 11
 10 11 12 14 4 15.3 7 9 10 11 11 12 14 4 15.2 7 9 9 10
 11 11 11 12 14 4 15.1 7 9 9 10 11 11 11 12 14 4 15.05
 7 9 10 11 11 12 14 4 15.03 7 9 9 10 11 11 11 12 14 4
 15.02 7 9 9 9 10 11 11 11 11 12 14 4 15.3 7 9 10 11 11
 12 15 4 15.2 6
A


•
•
•display 15-16
   15. (X1 ?) (CONT) => (REP X1 (0 X1)) (REM (CONT))
 15.02. (1 A ?) (2 A ?) (3 C ?) => (USED)
        (DEP (NEXT B))
 15.03. (1 A ?) (2 C ?) => (USED) (DEP (NEXT A))
 15.05. (1 C ?) => (USED) (DEP (NEXT A))
  15.1. (1 A ?) (2 A ?) => (USED) (DEP (NEXT C))
  15.2. (1 A ?) (2 B ?) => (USED) (DEP (NEXT A))
  15.3. (1 B ?) => (USED) (DEP (NEXT A))
  15.5. (1 A ?) => (USED) (DEP (NEXT B))
   16. (1 X4 ?) => (DEP (NEXT A))

•fire
 9 9 10 11 11 11 12 15 4 15.1 6
C

•fire
 9 9 10 11 11 11 12 15 4 15.05 6
A
```

APPENDIX VII. Trace of Complex Series Completion Production System.

```
•memory display ps display
MEMORY MODE
    1. STM • (READY)

PS MODE
    1  (READY) (SERIES X1)  •>  (REP READY CONT)
       (DEP (PNUM 2) (COUNTS 0)) (OBSERVES X1 ?)
    2  (READY)  •>  (ATTEND SERIES) (DEP (PERIOD 1))
    3. (COUNT) (COUNTS X1)  •>  (REM (COUNT))
       (REP X1 X1')
    4  (0 X1 ?)  •>  (SUCC)
    5  (FAIL) (PERIOD X1) (SERIES X2)  •>  (ERASE)
       (CLEAR) (DEP (READY) (PERIOD X1') (SERIES X2))
    6. (PERIOD X1) (COUNTS X1') (SERIES X2)  •>
       (ERASE) (CLEAR)
       (DEP (READY) (PERIOD X1') (SERIES X2))
    7. (NEXT X1) - (X2 ?) - (ACTION)  •>  (SAY X1)
       (DEP (MATCH) (X1 ?)) (STOP)
    8  (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
       - (MATCH) - (ERROR)  •>  (DEP (MATCH))
    9  (X1 X2 ?) (NEXT) - (DONE)  •>
       (DEP (OLD (X1 X2 ?))) (DEP (DONE))
   10. (USED (X1 X2 ?))  •>  (REP USED OLD)
       (DEP (X1 X2 ?))
   11. (OLD (X1 X2 ?))  •>  (REP OLD COND)
   12  (MATCH) (NEXT X1) (SERIES X2) (LOC X3)  •>
       (REP (NEXT X1) CONT 2)
       (REM (MATCH) (DONE) (LOC X3)) (PROD (SERIES X2))
   13  (LOC X1) (NEXT X2) (PERIOD X3) (SERIES X4)
       (COUNTS X5)  •>  (REM (LOC X1) (DONE) (ERROR))
       (REP (NEXT X2) CONT) (PRODS (LOC X1) X3 X4 X5)
   14. (CONT) (X1 ?) (PNUM X2) (X3 ?)  •>
       (REP X1 (0 X1) 2) (REP X2 X2' 3) (REM (CONT))
       (ACTION (USED) (DEP (NEXT X3)) (DEP (LOC X2)))
   15  (CONT) (X1 ?)  •>  (REP X1 (0 X1) 2)
       (REM (CONT))
   16  (1 X1 ?)  •>  (DEP (NEXT X1) (LOC 1))

•
•fire
 2
OUTPUT FOR (ATTEND SERIES) • (dep (series cdcdcd))
 1 14 4 16 9 11 13
NOW INSERTING
(1 X1 ?)  •>  (USED) (DEP (NEXT X1')) (DEP (LOC 2))
ON LINE  15.5
 3 14 4 15.5 9 10 11 11 13
NOW INSERTING
(2 X1 ?) (1 X2 ?)  •>  (USED) (DEP (NEXT C)) (DEP (LOC 3))
ON LINE  15.3
 5 1 14 4 16 9 11 13
NOW INSERTING
(1 X1 ?)  •>  (USED) (DEP (NEXT D)) (DEP (LOC 2)) (DEP (ERROR))
ON LINE  15.5
 14 4 15.5 9 10 11 11 13
NOW INSERTING
(2 X1 ?) (1 X2 ?)  •>  (USED) (DEP (NEXT X1)) (DEP (LOC 3))
ON LINE  15.3
 3 14 4 15.3 8 9 10 10 11 11 11 12 14 4 15.3 8 9 10 10 11
 11 11 12 14 4 15.3 8 9 10 10 11 11 11 12 15 4 15.3 7
C

•fire
 9 10 10 11 11 11 12 15 4 15.3 7
D
```

```
•display 15-16
   15  (CONT) (X1 ?)  •>  (REP X1 (0 X1) 2)
       (REM (CONT))
 15.3  (2 X1 ?) (1 X2 ?)  •>  (USED) (DEP (NEXT X1))
       (DEP (LOC 3))
 15.5  (1 X1 ?)  •>  (USED) (DEP (NEXT D))
       (DEP (LOC 2)) (DEP (ERROR))
   16  (1 X1 ?)  •>  (DEP (NEXT X1) (LOC 1))

•memory display ps display
MEMORY MODE
    1. STM • (READY)

PS MODE
    1  (READY) (SERIES X1)  •>  (REP READY CONT)
       (DEP (PNUM 2) (COUNTS 0)) (OBSERVES X1 ?)
    2  (READY)  •>  (ATTEND SERIES) (DEP (PERIOD 1))
    3  (COUNT) (COUNTS X1)  •>  (REM (COUNT))
       (REP X1 X1')
    4. (0 X1 ?)  •>  (SUCC)
    5. (FAIL) (PERIOD X1) (SERIES X2)  •>  (ERASE)
       (CLEAR) (DEP (READY) (PERIOD X1') (SERIES X2))
    6  (PERIOD X1) (COUNTS X1') (SERIES X2)  •>
       (ERASE) (CLEAR)
       (DEP (READY) (PERIOD X1') (SERIES X2))
    7  (NEXT X1) - (X2 ?) - (ACTION)  •>  (SAY X1)
       (DEP (MATCH) (X1 ?)) (STOP)
    8  (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
       - (MATCH) - (ERROR)  •>  (DEP (MATCH))
    9  (X1 X2 ?) (NEXT) - (DONE)  •>
       (DEP (OLD (X1 X2 ?))) (DEP (DONE))
   10  (USED (X1 X2 ?))  •>  (REP USED OLD)
       (DEP (X1 X2 ?))
   11  (OLD (X1 X2 ?))  •>  (REP OLD COND)
   12  (MATCH) (NEXT X1) (SERIES X2) (LOC X3)  •>
       (REP (NEXT X1) CONT 2)
       (REM (MATCH) (DONE) (LOC X3)) (PROD (SERIES X2))
   13  (LOC X1) (NEXT X2) (PERIOD X3) (SERIES X4)
       (COUNTS X5)  •>  (REM (LOC X1) (DONE) (ERROR))
       (REP (NEXT X2) CONT) (PRODS (LOC X1) X3 X4 X5)
   14  (CONT) (X1 ?) (PNUM X2) (X3 ?)  •>
       (REP X1 (0 X1) 2) (REP X2 X2' 3) (REM (CONT))
       (ACTION (USED) (DEP (NEXT X3)) (DEP (LOC X2)))
   15  (CONT) (X1 ?)  •>  (REP X1 (0 X1) 2)
       (REM (CONT))
   16  (1 X1 ?)  •>  (DEP (NEXT X1) (LOC 1))

•
•fire
 2
OUTPUT FOR (ATTEND SERIES) • (dep (series ebmcdmef))
 1 14 4 16 9 11 13
NOW INSERTING
(1 X1 ?)  •>  (USED) (DEP (NEXT X1')) (DEP (LOC 2))
ON LINE  15.5
 3 14 4 15.5 9 10 11 11 13
```

```
NOW INSERTING
(2 X1 ?) (1 X2 ?) => (USED) (DEP (NEXT M)) (DEP (LOC 3))
ON LINE 15.3
 5 1 14 4 16 9 11 13
NOW INSERTING
(1 X1 ?) => (USED) (DEP (NEXT B)) (DEP (LOC 2)) (DEP (ERROR))
ON LINE 15.5
 14 4 15.5 9 10 11 11 13
NOW INSERTING
(2 X1 ?) (1 X2 ?) => (USED) (DEP (NEXT M)) (DEP (LOC 3))
ON LINE 15.3
 5 1 14 4 16 9 11 13
NOW INSERTING
(1 X1 ?) => (USED) (DEP (NEXT B)) (DEP (LOC 2)) (DEP (ERROR))
ON LINE 15.5
 14 4 15.5 9 10 11 11 13
NOW INSERTING
(2 X1 ?) (1 X2 ?) => (USED) (DEP (NEXT M)) (DEP (LOC 3)) (DEP (ERROR))
ON LINE 15.3
 14 4 15.3 9 10 10 11 11 11 13
NOW INSERTING
(3 X1 ?) (2 X2 ?) (1 X3 ?) => (USED) (DEP (NEXT X1")) (DEP (LOC 4))
ON LINE 15.2
 3 14 4 15.2 8 9 10 10 10 11 11 11 11 12 14 4 15.2 9 10 10 10 11 11 11 11 13
NOW INSERTING
(4 X1 ?) (3 M ?) (2 X3 ?) (1 X1" ?) => (USED) (DEP (NEXT M)) (DEP (LOC 6))
ON LINE 15.1
 3 14 4 15.2 8 9 10 10 10 11 11 11 11 12 14 4 15.2 8 9 10 10 10 11 11 11
 11 12 15 4 15.1 7
M

•fire
 9 10 10 10 10 11 11 11 11 11 12 15 4 15.2 7
G

•fire
 9 10 10 10 11 11 11 11 12 15 4 15.2 7
H

•display 15-16
   15. (CONT) (X1 ?) => (REP X1 (0 X1) 2)
       (REM (CONT))
 15.1. (4 X1 ?) (3 M ?) (2 X3 ?) (1 X1" ?) => (USED)
       (DEP (NEXT M)) (DEP (LOC 6))
 15.2. (3 X1 ?) (2 X2 ?) (1 X3 ?) => (USED)
       (DEP (NEXT X1")) (DEP (LOC 4))
 15.3. (2 X1 ?) (1 X2 ?) => (USED) (DEP (NEXT M))
       (DEP (LOC 3)) (DEP (ERROR))
 15.5. (1 X1 ?) => (USED) (DEP (NEXT B))
       (DEP (LOC 2)) (DEP (ERROR))
   16. (1 X1 ?) => (DEP (NEXT X1) (LOC 1))
```